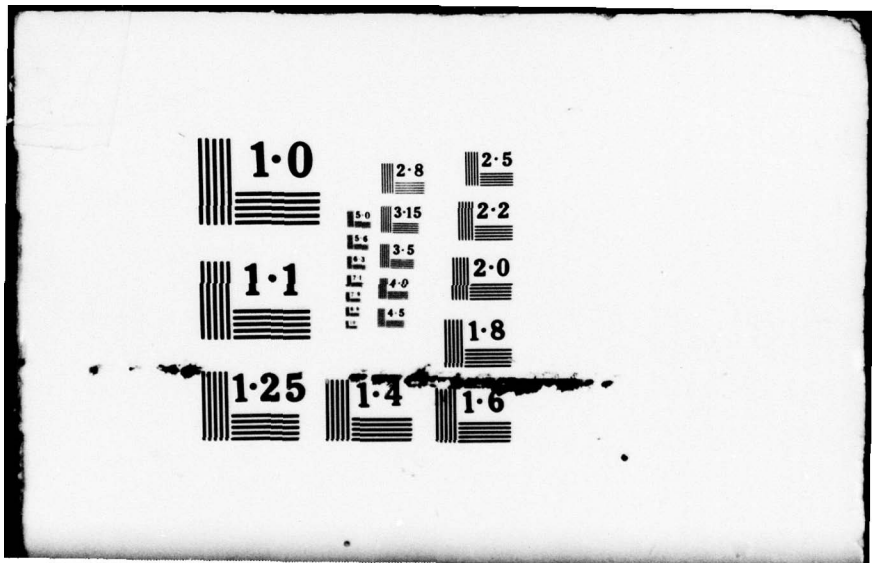
AD-A065 193 ARIZONA UNIV TUCSON DEPT OF COMPUTER SCIENCE F/G 9/2
RELIABLE SOFTWARE THROUGH VERY HIGH-LEVEL VERIFICATION.(U)
OCT 78 R J ORGASS AFOSR-78-3499
UNCLASSIFIED TR-APLAD15 AFOSR-TR-79-0032 NL

END
DATE FILMED
4-79
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0

AFOSR-TR- 79-0032

LEVEL II

AD A065193

DDC FILE COPY

Department

of

Computer Science

79 02 15 008

The University of Arizona

Approved for public release;
distribution unlimited.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-79-0032	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): RELIABLE SOFTWARE THROUGH VERY HIGH-LEVEL VERIFICATION.	5. TYPE OF REPORT & PERIOD COVERED: Final rept.,
6. PERFORMING ORG. REPORT NUMBER: TR-APLAD15	
7. AUTHOR(s): Richard J. Orgass	8. CONTRACT OR GRANT NUMBER(s): AFOSR-78-3499
9. PERFORMING ORGANIZATION NAME AND ADDRESS: The University of Arizona, Department of Computer Science, Tucson, Arizona 85721	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332	12. REPORT DATE: October 10, 1978
	13. NUMBER OF PAGES: 80 82 p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A semantics of a significant fragment of APL has been constructed and is summarized in Section 1. This semantics has been used to construct an implementation of the fragment of APL and part of the implementation has been verified. This work is summarized in Section 2.

This research program has been moved from the University of Arizona to Virginia Polytechnic Institute and State University. As a result of this move, the computing services used for this software have been changed

A065 127

DD FORM 1 JAN 73 1473 A

411088

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20 Abstract continued.

from a DECsystem-10 to an IBM system 370/158. Although there were some difficulties, that concern the physical representation of the programs, this experience justifies the claim that the software is substantially machine independent. The current state of the soft-ware and some of the problems encountered in the move are described ~~in Section 5.~~

The starting point for the construction of an APL verifier is an incremental assertion synthesizer that was written at the Univ~~ersity~~ of Arizona, as the dissertation of Dr. Dianne E. Britton.

The next step in the research is to complete the verification of the first part of the APL implementation. When this is completed, the work will branch into two cooperating projects. One of these projects is to complete the semantics and implementation of APL. The other is to derive the rules of inference for the APL primitives that have been defined and to work with the verifier to extend it to a more powerful verifier. After the semantics of APL are completed, the verifier will be further extended.

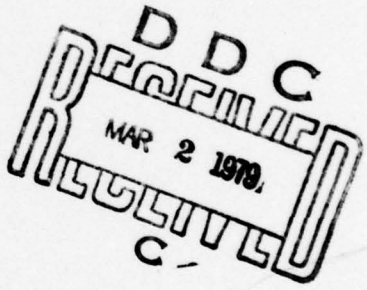
FINAL SCIENTIFIC REPORT

Grant AFOSR-78-3499

Richard J. Orgass

October 10, 1978

Technical Report No. APLAD15



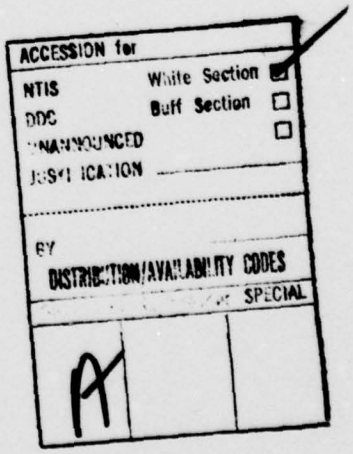
Submitted to

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH
Building 410
Bolling Air Force Base
Washington, D. C. 20332

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

UNIVERSITY OF ARIZONA
Department of Computer Science
Tucson, Arizona 85721

TABLE OF CONTENTS



0. OVERVIEW

The overall objective of this research program is to provide a set of tools that can be used to verfiy substantial APL programs and to provide a verified implementation of APL that is capable of correctly executing verified APL programs. Before the grant began a prototype verifier was constructed and this work defines the direction of verification work. Continued work on verification depends on the availability of a formal semantics of APL and, therefore, during this research period attention was directed to work on the semantics and implementation of APL.

A semantics of a significant fragment of APL has been constructed. This work is summarized in Section 1, below. This semantics has been used to construct an implementation of the fragment of APL and part of the implementation has been verified. This work is summarized in Section 2 below.

This research program has been moved from the University of Arizona to Virginia Polytechnic Institute and State University. As a result of this move, the computing services used for the software have been changed from a DECsystem-10 to an IBM System 370/158. Although there were some difficulties that concern the physical representation of the programs, this experience justifies the claim that the software is substantially machine independent. The current state of the software and some of the problems encountered in the move are described in Section 5.

It was necessary to do some technical work to accomplish the program transfer. The work on the transfer of APL workspaces, summarized in Section 4, below, is a contribution to the general problem of transferring APL workspaces across implementations. In addition, some programs to transfer files from a DEC-10 to an IBM machine were written (Section 3). These programs were also used to transfer software from the University of Arizona to RCA Laboratories by two faculty members who moved to the Laboratories.

The starting point for the construction of an APL verifier is an incremental assertion synthesizer that was written at the University of Arizona as the dissertation of Dr. Dianne E. Britton. This program and its supporting documentation have been moved to Virginia Tech and it appears to be working correctly.

The next step in the research is to complete the verification of the first part of the APL implementation. When this is completed, the work will branch into two cooperating projects. One of these projects is to complete the semantics and implementation of APL. The other is to derive the rules of inference for the APL primitives that have been defined and to work with the verifier to extend it to a more powerful verifier. After the semantics of APL are completed, the verifier will be further extended.

1. SEMANTICS OF APL

Substantial progress has been made in constructing a primitive recursive semantics of APL. In particular, a semantics of essentially all of the primitive scalar functions as well as simple and subscripted assignment and a basic set of mixed functions has been constructed. Perhaps more important, an orderly framework for constructing a semantics of the entire language as well as other languages has been constructed.

The results of this work are described in the first two chapters of a research monograph "A Primitive Recursive Semantics and Implementation of APL". These two chapters have been issued as technical reports using grant funds. These reports are Attachment 1 to the present report and the external distribution list is Attachment 2. Completed work as well as plans for completing the semantics of APL are summarized below. Work on a verified implementation of APL based on this semantics is described in Section 2, below.

The basic assumption of this work is that the semantics of a programming language is given by exhibiting a valuation function that maps well-formed expressions or statements of the language into their values or meanings. In the present work, the well-formed expressions are the expressions of APL as the language is used in a number of implementations. The values of individual syntactical objects (identifiers, primitive function symbols, etc.) are defined to be specific objects in some mathematical system. With these definitions in hand, the values of expressions are defined in terms of the meanings of the syntactical objects that occur in the expressions. Primitive recursive arithmetic was selected as the mathematical system for the meanings because it has a well developed proof structure and is adequate for the present purposes. A more detailed justification for this choice appears in Chapter I of the manuscript.

The fundamental data objects of APL, called APL individuals, are scalars and arrays of type number, boolean and character. Since the domain of individuals of primitive recursive arithmetic is natural numbers, these APL individuals are mapped into natural numbers using pairing functions. The set of natural numbers that correspond to APL individuals is the domain of the primitive recursive functions that are the meanings of the primitive functions of APL. Although the correspondence between APL individuals and natural numbers is defined in terms of pairing functions, the details of the representation are such that it is very straightforward to implement APL individuals from the definition of the mapping.

It is, in principle, possible to exhibit a primitive recursive function that is the meaning of each of the primitive function symbols of APL. However, for a number of reasons, it is desirable to directly define a limited set of primitive functions and this limited set of

primitive functions is the subject of Chapter II of the manuscript.

The main motivation for directly defining only a limited set of primitive functions is economy of definition and proof. Primitive recursive definitions, even with a number of functions that facilitate the definition of APL primitive functions, are quite long and complicated. Once a reasonable set of APL primitive functions has been defined in recursive arithmetic, these functions can be used to define additional functions. The definitions of these functions in terms of earlier functions are significantly shorter and, therefore, are easier to understand and to confirm by comparison with existing implementations. When the APL program verifier is constructed, rules of inference for each primitive function will be derived from the definitions. The functions defined in terms of other functions have the property that the derivation of their rules of inference will be shorter because more powerful rules of inference are available.

The limited set of primitive functions directly defined in Chapter II includes all of the primitive scalar functions of APL except roll, deal (monadic and dyadic ?) and the transcendental or circular functions (dyadic ○). The remaining primitive scalar functions were included because their definition fits one of three patterns, one for all of the manadic scalar functions and two for the dyadic scalar functions. The mixed functions that are included are shape and reshape (monadic and dyadic ⍴), ravel and catenate (monadic and dyadic ,), reverse (monadic ⌽), subscripting ([]), index generator (monadic ⍳), membership (dyadic ∈), simple and subscripted assignment (←) as well as two supporting functions whose value is the cardinality and type of their arguments. It is fairly straightforward to show that a subset of these functions is sufficient to define all of APL but this proof is not directly relevant to the present research because our interest is in providing a definition that is easy to work with.

A brief discussion of the structure of the semantics of APL as constructed in this work may be helpful.

A valuation function maps each identifier and each primitive function symbol into its value or meaning. Most of the valuation functions that have the set of APL expressions as their domain are not of interest here because they do not assign the appropriate value to expressions and, therefore, attention is restricted to admissable valuation functions. Each admissable valuation function assigns the appropriate function as the value of a primitive function symbol and an APL individual as the value of an identifier. This APL individual may be an ordinary data object (a scalar or array of type number, boolean or character), a label or a function definition. Thus, admissable valuation functions may only differ in the values that they assign to identifiers.

At a particular time, the state of an APL workspace is given by the current valuation function. [This statement assumes that system variables as defined in APL.SV, APLSF and other modern implementations are available.] The state of a workspace is changed by executing an

assignment, calling a function and by completing the execution of a line of a function. This state change corresponds to changing the valuation function associated with the workspace.

The semantics of the state changing operations is considerably simplified if the valuation function itself is available in the mathematical system used for the semantics as an object that is easily changed. The valuation functions that are used in the present semantics are defined within primitive recursive arithmetic.

This definition is completed in two essentially independent steps. Strings of APL characters are not included in the domain of individuals of recursive arithmetic but it is quite easy to represent them as such. An APL individual is said to contain a string of APL characters just in case it is a vector of type character such that each component of the vector is the representation of the corresponding character of the character string. This mapping of strings of APL characters into APL individuals that contain the strings is, in fact, a Gödel numbering of the strings and the domain of the valuation functions that are used is the set of APL individuals that contain strings of APL characters. With this change in point of view, both the domain and the range of valuation functions are objects of primitive recursive arithmetic.

The definition of a valuation function may be divided into two parts, a fixed part that is common to all valuation functions and a variable part that is different for each valuation function. The fixed part of the definition of a valuation function is the part that assigns values to the primitive function symbols of the language and defines the value of expressions in terms of the values of constituents of the expressions. The variable part of the definition is the part that assigns values to identifiers.

The variable part of a valuation function is defined by a symbol table. A symbol table is a linear list of ordered pairs of APL individuals. Each of these pairs consists of a left part and a right part. The left part is an APL individual that contains an identifier and the right part is the APL individual that is the value of the identifier. In addition, each symbol table is required to define certain system identifiers. For example, in the first level semantics (Chapter II), the system variables $\square PP$, $\square PW$, $\square IO$ and $\square CT$ must be defined. In later extensions, additional system variables are added.

Restricted expressions are expressions that do not contain occurrences of the assignment function symbol (←). The valuation function for restricted expressions is defined by means of a function, V, of two arguments. The first argument is an APL individual that contains an APL expression and the second argument is a symbol table. The value of this function V is the value of the expression when the identifiers have the values given in the symbol table. It has been shown that for each admissable valuation function there is a symbol

table $\underset{\sim}{t}$ such that $V(X,\underset{\sim}{t})$ is the value that the valuation function assigns to the expression contained in X. Furthermore, for each symbol table $\underset{\sim}{t}$, there is an admissable valuation function that assigns the value $V(X,\underset{\sim}{t})$ to the expression contained in X. Therefore, the two are equivalent definitions of valuation functions.

The set of simple expressions, whose semantics is given in Chapter II is the set of restricted expressions with both simple and vector assignment added. A function $\underset{\sim\sim}{apl}$ whose argument is an ordered pair, $\langle X,\underset{\sim}{t}\rangle$, where X is an APL individual that contains a simple expression and $\underset{\sim}{t}$ is a symbol table and whose value is a second ordered pair, $\langle Y,\underset{\sim}{t}_1\rangle$ such that Y is the value of the expression contained in X and $\underset{\sim}{t}_1$ is the symbol table after the expression is evaluated. This definition is adequate to give a complete semantics of this fragment of APL and to construct an implementation of APL as described below.

The manuscript describing this semantics has been distributed to a number of people that are actively engaged in constructing or maintaining APL implementations and to others who are interested in the semantics of programming languages. [The distribution list is Attachment 2 to this report.] Comments received to date indicate that there is substantial agreement with this definition of APL. There is, however, one area of disagreement. In APLSF (DEC's APL), the comparison tolerance ($\square CT$) and fuzz are the same. That is, if a real is within $\square CT$ of an integer, it is treated as an integer. In the IBM implementations, $\square CT$ is a relative error when two reals are compared and a real is treated as an integer if it becomes an integer when a number of low order bits named by fuzz are set to 0. A satisfactory resolution of this difference has not been identified at this time.

The next step in the work on the semantics of APL is to define the semantics of user defined functions. This will be done by defining the value of a function variable in terms of an algorithm for computing the value of the function and executing this algorithm. This definition requires the addition of a few more details to the structure of the semantics but no fundamental changes are anticipated. This work is to be Chapter IV of the monograph mentioned above.

After the semantics of user defined functions is completed, the remaining work on the semantics is to define the remaining primitive functions of APL in terms of the functions that have been defined. These function definitions are added to the symbol table and the result is a complete semantics of APL.

This work on the semantics of APL is closely related to the implementation work described in Section 2, below. Chapters in the research monograph alternate between describing the semantics and verifying that the implementation is correct with respect to the semantics.

2. A VERIFIED IMPLEMENTATION OF APL

A SIMULA-67 implementation of the APL fragment described above has been constructed and a portion of this implementation has been verified with respect to the semantics. The implementation has been transported from a DECsystem-10 to an IBM System 370/158 and this move provides considerable support for the claim that it is machine independent. [The major difficulties associated with the move concerned the physical representation of the program text not the text itself.]

This implementation is to be described in the third Chapter of the research monograph "A Primitive Recursive Semantics and Implementation of APL". This chapter is, at present, incomplete and the existing fragment of the manuscript is Appendix I to the present report. The following remarks provide an overview of the implementation and its verification.

The implementation was constructed to closely follow the structure of the semantics and this decision greatly simplified both the construction and the verification of the implementation. It is interesting to note that questions that arose in the construction of the implementation often motivated simplifications in the semantics.

The original plan for this work called for verification of the program while it was being constructed. This plan was not followed and verification began after the program appeared to be working correctly. It was a pleasant surprise to find that construction of a proof of correctness suggested program changes that both simplified and improved the efficiency of the program. In addition, coding errors that were not detected by extensive testing were detected and corrected while constructing an informal proof of correctness.

The most interesting discovery of this work is that simply constructing an implementation that closely follows a precise mathematical definition does not guarantee a simple verification of the implementation. Some very interesting questions about writing the specifications of a program must be dealt with in a satisfactory way.

As an example, consider the implementation of APL individuals. The set of APL individuals in the formal semantics is an inductively defined set of n-tuples of natural numbers. This is a highly structured definition and the abstraction mechanism of SIMULA is sufficiently powerful to deal with this kind of definition in a straightforward way. Indeed, the SIMULA definition of the class *apl_individual* that appears in Section 31.2 of Appendix I is, in many ways, easier to understand than the definition given in Section 21.4 of Attachment 1. Is the set of APL individuals the same set of objects as the set of instances of class *apl_individual?* Intuitively, the answer is "yes,

with some minor qualifications". Establishing this statement with some degree of precision is rather more difficult!

Some of the components of APL individuals are rationals and only a subset of the rationals can be represented using the floating point hardware of the host computing machine. This problem has been avoided in the present work by assuming that the host machine adequately approximates rationals by selecting an appropriate precision for reals. This issue is of importance but is beyond the scope of the present research.

The finite word length of the host machine also restricts the magnitude of integers. This imposes limits on the rank, shape and cardinality of APL individuals in the implementation. Let *maxint* be the largest integer that can be represented on the host machine. We would like to prove two theorems:

Theorem 1. Each APL individual such that its rank and cardinality are less than or equal to *maxint* and such that each component of its shape is less than or equal to *maxint* is implemented by an instance of class *apl_individual*.

Theorem 2. Each instance of class *apl_individual* implements a member of the set of APL individuals.

The first theorem is fairly easy to prove once the definition of implementation has been given. This definition essentially asserts that there is a correspondence between the attributes of an APL individual and the attributes of the implementation. The exact statement of this definition is a fairly delicate problem and sharpened this writer's understanding of implementation.

The second theorem is considerably harder to prove and the proof of this theorem motivated a substantial change in the original definition of class *apl_individual*. Ideally, one would like to prove that each instance of this class has all of the attributes of an APL individual. To prove this, one must show that it is not possible to create an instance of this class that does not satisfy this definition and that once an instance of the class is created it cannot be changed to an object that does not satisfy the definition of an APL individual.

The protection mechanism of SIMULA plays an essential part in the proof of this theorem to the extent that it can be proved. It is possible to provide read-only access to class attributes and this makes it possible to prohibit changes to some attributes. However, it is necessary to change other attributes if the implemented individual is to be useful. Even with the help of these protection mechanisms, it is only possible to prove the following theorem:

Theorem 2a. As long as execution of the implementation continues without error termination, each instance of class *apl_individual* implements an APL individual.

Even with a proof of this theorem in hand, one is left with a rather insecure view of the implementation. When will it error terminate without warning? This insecurity is relieved to some extent by a theorem which asserts that error termination in class *apl_individual* will occur in a limited number of circumstances that are enumerated in the theorem.

This alone does not provide much added confidence. In addition, it must be shown that each time one of these circumstances could arise, the program that manipulates APL individuals has the property that it cannot arise. This adds a significant amount of work to the problem of proving that the implementation is correct.

Stated in another way, it is necessary to prove two properties of the implementation: (1) As long as the program is running without error termination, it is a correct implementation of APL as defined in the semantics. (2) The program will not have an error termination.

It is not necessary to prove termination in the usual sense of this term for the following reason. An APL interpreter continues to interpret input lines one after the other without ever terminating execution. There is one distinguished input which causes program exit: *)OFF*. It must be shown that this input will cause program exit.

It is necessary to prove a theorem that asserts that the interpreter will correctly interpret any input line and that after this line has been interpreted, it will again correctly interpret an input line.

This discussion of the verification of the implementation is incomplete because all of the issues to be addressed have not yet been considered. They will be addressed as part of continuing work on this research under grant AFOSR-79-0021.

The implementation is not described in any grant publications and it may be helpful to provide an overview of the implementation. The program consists of about 3250 lines of SIMULA code including many blank lines to improve readability.

The program is divided into ten layers. Each layer has access to all of the objects declared at lower levels. This structure was selected to reduce the possibility of incorrect interactions. The program structure alone does not prohibit all incorrect interactions and those which are not precluded by the program structure must be shown to be absent.

The bottom layer of the program is class *apl0* which contains functions and predicates that are used in all higher levels. These procedures include input/output translations, the fuzzy comparisons and data structures that are used at higher levels. When this class instance is created, a variety of initializations are performed.

The next layer is class *apl1* and this is where the class *apl_individual* discussed in Section 31.2 of Appendix I is declared. In addition, procedures that implement the predicates of APL individuals that are defined in Section B5 of the research monograph are declared. These procedures are to be verified in appendicies of the research monograph. In addition, the distinguished APL individuals are declared.

The next two layers, classes *apl2* and *apl3* contain declarations of the primitive function classes. The first of these classes is the *primitive_function* class *monadic_scalar*. Recall that the monadic scalar functions are defined using a template with three blanks. This template is implemented in class *monadic_scalar*. The blanks are filled in using subclasses of *monadic_scalar* which define the missing pieces. One instance of each of the subclasses is to be created and placed in the primitive function table. Each subclass has a procedure attribute *perform* and this procedure is executed to execute the primitive function. Similar statements apply to the two dyadic function templates; they are implemented as classes *dyadic_1* and *dyadic_2*.

The mixed function definitions do not have a clearly visible structure and they are defined in a more or less independent way but references to these functions are dealt with in the same way.

From a vantage point just above class *apl3*, the set of APL individuals and the primitive functions defined in the semantics, except for assignment, are available.

Class *apl4* is, in some ways, a transition between the purely semantic objects below and the mixture of semantic and syntactical objects above. Some components of the syntax of APL are defined and symbol tables and functions on symbol tables that are defined in the semantics. These objects are used to implement assignment in the procedure *assign*. This matches the definition of assignment given in the formal semantics.

Class *apl5* contains the declaration of some of the components of the syntax analyzer. These include some of the procedures used in the syntax analyzer. In addition, the correspondence between primitive function symbols and procedures that implement them are declared and initialized. This class is best viewed as creating the working environment for the syntax analyzer.

Class *apl6* is the syntax analyzer. This program is basically a finite state machine parser that accepts an input line (as an *apl_individual* that contains the string) and constructs a syntax tree which is traversed by the interpreter.

Class *apl7* contains the expression evaluator. The class declaration includes declarations of all of the procedures that implement the functions that are used to define the function apl in the semantics as well as the implementation of apl itself.

The next layer, class *apl8*, contains initializations of a number of constants that are used in the main program.

The top layer is the main program itself. It begins by performing a number of initializations that are required at lower levels. The main part of this program consists of a loop that examines an input line by calling the syntax analyzer and then applying the interpreter to this line. A number of global error checks that are related to the input/output operations are performed. In addition, some error checking that arises because the entire language is not yet implemented is performed. This program will be replaced in the final implementation.

The plan for extending this implementation to a complete APL implementation may be summarized as follows. After the semantics of function execution is given, the interpreter function will be extended to include execution of user defined functions. The semantics will include an APL defintiion of the remaining primitive functions and these definitions will be added to the initial symbol table. When this is done, the language implementation will be complete except for a few "utility" procedures.

The symbol table completely characterizes an APL workspace. In order to implement the system command *)SAVE*, a symbolic representation of the symbol table will be selected and used to write workspaces on disk files. The inverse of *)SAVE*, namely *)LOAD*, will also be written. Lastly, *)CONTINUE*, a combination of *)SAVE* and *)OFF* will be written. These system commands are not a part of the semantics and are viewed as utilities. The remaining system commands are, in fact, duplicated by system functions and are part of the formal definition of APL.

3. TRANSFER OF PROGRAM AND DATA FILES TO VIRGINIA TECH

The most difficult and frustrating task associated with moving this research from the University of Arizona to Virginia Polytechnic Institute and State University was the transfer of data files from the DECsystem-10 at the University of Arizona to the IBM System 370/158 at Virginia Tech. Once the files were transferred, it was fairly easy to modify SIMULA programs for execution using the compiler written for IBM machines by the Norwegian Computing Center.

Commercial computer service organizations are well prepared to receive data files from other installations and to transmit files to outside machines. In contrast, University Computer Centers provide limited help for this kind of file transfer. The staff of the University of Arizona Computer Center provided technical advice and the Department of Computer Science at Virginia Tech provided substantial technical staff support to complete the file transfer. Without this support from Virginia Tech, the file transfer would not have been successful.

The technical problems associated with the file transfer are best described as a sequence of annoying problems concerning the obscure details of tape representations, etc.

The first step was to find a tape format that could be written by a DEC-10 and read by an IBM machine. This format is blocked, fixed-length records without labels of any kind in either ASCII or EBCDIC characters. A program to write such tapes was found in the UA program library. The documentation for this program was rather sketchy and three attempted interchanges were attempted before the fourth successful interchange. A CMS EXEC procedure to generate batch jobs to selectively load DEC-10 files onto an IBM machine was written. This tape load procedure has been found to be very reliable and the DEC-10 tapes are now used for off-line storage of files.

The second step was to examine each DEC-10 file to determine file descriptions that are required by IBM machines. For a DEC-10, a file is a sequence of ASCII characters divided into records by the character sequence carriage-return, linefeed and the number of records in the file as well as the length of the longest record is not known. When writing files for an IBM machine, the number of records in the file and the length of the longest record must be known so that fixed length records can be written on tape and then read from tape. A program to examine files and collect the required data was written. The report produced by this program is processed by a second program that generates commands for the tape writing utility. A listing of this command file also provides information that is needed when loading files onto IBM machines.

The third step was to assemble all of the files to be transferred on disk and to make a number of changes in the files before

writing them on transfer tapes. These changes include deleting line numbers and page marks as well as replacing tabs with the appropriate number of blanks. In addition, some language specific changes, described in Section 5, were made. This step consumed considerable time because many files were stored on DEC-10 BACKUP tapes.

The fourth and final step was to read the files onto the disks of the IBM machine and recompile the programs. The status of this work is described in Section 5.

There are two specific accomplishments to report:

(1) All relevant files were successfully transferred from the University of Arizona to Virginia Tech.

(2) A technical memorandum describing the file transfer procedure was written and distributed to the Departments of Computer Science and Computer Centers of Virginia Tech and the University of Arizona. A copy of this memo (Technical Memorandum No. APLAD14) is Attachment 3 to this report.

4. TRANSFER OF APL WORKSPACES

Some of the ideas used in an earlier set of workspaces to verify APL programs [Feldbrugge, 1973a, 1973b] are relevant to the present research and it has been helpful to have these workspaces available. In addition, a number of small workspaces that support the present research have been written and remain useful. These workspaces have been transferred from the DECsystem-10 at the University of Arizona to the IBM System 370/158 at Virginia Polytechnic Institute and State University. The status of these workspaces is summarized in Section 5.

The transfer of APL workspaces is more complicated than the transfer of source program files for compiled languages because APL workspaces are stored as binary files on disk and this format is completely different for different implementations. Some character representation of the workspace is required for transmission from one computer to another.

This kind of workspace interchange will be quite easy when the Proposed STAPL Convention for the Interchange of APL Workspaces [APL Quote Quad, Fall 1977, pp. 25-35] is adopted and implemented by all implementors of APL. Since this software was not available, two representations of APL workspaces as character files were implemented. One of these is a simple terminal transcript with the property that when the text of the file is read by an APL interpreter the workspace is recreated. The second representation conforms to the STAPL convention.

There are a number of incompatibilities between all of the existing APL systems. The main source of these incompatibilities is the definition of file input/output and system variables. These differences also introduce differences in the character set of the implementations so that a function definition that is syntactically acceptable in one implementation is unacceptable in another. A reasonably complete list of incompatibilities between Digital Equipment's APLSF and IBM's VS/APL was compiled and a set of directions for modifying workspaces to avoid these incompatibilities has been assembled.

A set of functions that can be used to write a terminal transcript of a workspace has been written and used to transfer APL workspaces from a DEC-10 to an IBM machine. These functions are read into a workspace that is to be transferred and when the functions are executed a terminal transcript is written as a key-paired ASCII disk file. When the APL interpreter is directed to read this file as though it were terminal input, the original workspace is recreated. Files written in this form are then transferred to another machine using the programs discussed in Section 3. This procedure has successfully transferred workspaces to the 370/158 at Virginia Tech.

The Proposed STAPL Convention for the Interchange of Workspaces defines three levels of workspace representation. In the first level, each object in the workspace, functions and variables, are represented as character vectors. There are simple functions for converting objects into character vectors and for reconstructing the objects given the character vectors. At the second level, these character representations of objects are combined into a single character string that represents the entire workspace. The functions for converting to this representation and for restoring the objects are more complicated but they still use the level one functions. In addition, a fixed format for these character strings as records in a file is defined. The third level convention establishes the exact format in which workspaces are to be written on nine track magnetic tape. This format is a string of bits which is presumably independent of any computer manufacturers hardware and which can be read and interpreted on any machine.

During the period of the grant, functions to create STAPL level 1 and level 2 representations of workspaces were written and used to write APL workspaces as disk files. These files were transferred to Virginia Tech and have been read and used to reconstruct workspaces. After the termination of this grant, using computing funds provided by the Department of Geosciences and the Computer Center at the University of Arizona, functions to create and read workspaces written using STAPL level 3 representations were written and tested. These functions have not been used across different machines but they are known to be inverses of each other on a DEC-10.

There is a second application for the STAPL level 2 representation of APL workspaces. The usual binary files for APL workspaces require a substantial amount of disk space. For a DEC-10, the level 2 character representation of a workspace requires about half as much disk storage. For infrequently used workspaces, this representation can reduce the cost of disk storage.

In summary, the following has been accomplished:

(1) APL workspaces used in this research as well as other workspaces were transferred from the DEC-10 at the University of Arizona to the IBM System 370/158 at Virginia Tech.

(2) APL functions to represent a workspace as a file that is a terminal transcript that will recreate the workspace were written and used to transfer workspaces.

(3) Functions to write STAPL Convention Level 2 representations of workspaces and to restore workspaces from this representation were written and used to transfer workspaces.

(4) Functions to write and read STAPL Convention Level 3 representations of workspaces were written and tested using the DEC-10 at the University of Arizona.

(5) A technical memorandum describing this work that includes directions for using the programs and the actual program text was written. This report is Attachment 4 to the present report.

5. STATUS OF PROGRAMS AT VIRGINIA TECH

The most important programs used in this research appear to be running correctly on the Virginia Tech IBM 370/158. The most difficult part of the conversion was preserving the interactive nature of the programs in spite of a much more limited host interactive system. It was a pleasant surprise to find that the SIMULA programs constructed as part of this research were the easiest programs to transport. Next in difficulty were the APL programs that support this work and the most difficult programs to move were the Fortran programs that support some of the SIMULA programs. The APL implementation described in Section 2, above, is now working correctly as is the interactive verifier described in [Britton, 1977]. The following paragraphs describe some of the difficulties that were encountered when transferring these programs.

There are minor differences in the details of terminal transcripts when running all of the programs used in this research. For example, DEC-10 SIMULA provides a procedure breakoutimage which transmits a line to the terminal and does not terminate this line with a carriage return-line feed character pair. Using this procedure, it is possible to prompt for a response and have the response entered on the same line as the prompt. This facility is not available in IBM SIMULA, and, therefore, prompts and responses are on different lines. In the IBM SIMULA implementation, the system procedure Outimage is buffered for one line. This means that an output line is retained in memory until the output line which succeeds it is created and ready for printing. This means that responses to user input are delayed by one line. The difficulties raised by this mode of operation can be eased by including additional blank lines in the terminal transcript using additional calls on the procedure Outimage.

The first program to be converted was the APL implementation described above and this conversion required approximately one man week. During this conversion work, some minor incompatibilities between the DEC-10 and IBM SIMULA implementations were discovered. In addition, one language incompatibility and one system error in the IBM implementation were discovered. These differences are summarized in the following paragraphs.

There are differences in the computer representation of SIMULA programs that are not fully described in the DEC-10 SIMULA documentation. The IBM compiler requires 80 column fixed length records and columns 73 to 80 are ignored because this is space for sequence numbers on cards. Although all of the programs had line lengths less than or equal to 80 characters, a substantial amount of editing to make each line 72 characters or fewer was required. Except in text constants, the IBM SIMULA compiler does not accept lower case letters. Using the DEC-10 implementation, which supports upper and lower case letters, a programming convention for the use of capital letters for key words, lower case letters for ordinary identifiers, and initial

letter capitalized words for system procedures was used. It was necessary to change all of these lower case letters to upper case in order to use the IBM implementation. Again, a substantial amount of editing was required so as to preserve lower case terminal prompts which are essential in the APL implementation.

A number of changes in the program text were required in order to preserve a readable listing. For example, the DEC-10 SIMULA compiler uses the character line feed to indicate the beginning of a new listing page and the character vertical table to indicate that lines are to be skipped in the output listing. These characters are not routinely supported by CMS and, therefore, they were removed from the program text prior to transmitting the programs to Virginia Tech. It was necessary to introduce %TITLE and %PAGE control cards as well as additional blank lines to recreate a readable listing. While there is nothing difficult about this task, it consumes a substantial amount of time.

One incompatibility between the DEC-10 and IBM SIMULA implementations was detected. This change, which required minor program modifications, may be described with the help of Figure 5-1. Observe that class A has a procedure attribute foo. This interger procedure returns a value 5. Class B is declared to be a sub-class of class A and uses a call on the procedure foo to fix the upper bound on the array elements. In the DEC-10 implementation, the creation of an instance of b will proceed correctly and the upper bound of the array elements will be 5. In contrast, the IBM compiler rejects this program text pointing out that the procedure foo is declared at the same level as the reference to foo. This diagnostic appears to be in variance with the definition of SIMULA given in the common base and has been reported to the Norwegian Computer Center.

```
begin
    class a;
    begin
        integer procedure foo;
            foo := 5;
            ...
    end;
    a class b;
    begin
        real array elements[1:foo];
        ...
    end;
    ref(b) x;
    x :- new b
end
```

Figure 5-1

There appears to be an error in the IBM SIMULA run time system. When executing the APL interpreter described above, an error message indicating that an actual parameter to a procedure is not of type numeric when, in fact, the parameter is a real procedure. It is possible to avoid this run time error by declaring a real variable and assigning the return value of the real procedure to this variable and then using the variable as an actual parameter to the procedure in question. This construction is somewhat less efficient and makes verification of the program slightly more complicated. A report of this error has been transmitted to the Norwegian Computer Center.

Using the information gained while converting the APL interpreter from DEC-10 SIMULA to IBM SIMULA, the program verification system described in [Britton, 1977] was converted to three man days. This program has been partially tested on the IBM system but not all paths of the program have been executed so there may be additional difficulties. The principal effort involved in converting this program was editing to shorten program lines to 72 characters and to modify text constants in order to preserve upper and lower case prompts after the program text was converted to upper case in order to satisfy the IBM compiler.

Forty-three APLSF workspaces were transferred from the University of Arizona to Virginia Tech. This set of workspaces includes workspaces used in the current research as well as administrative workspaces and other utilities that are useful in a variety of applications. The workspace transmission procedure described in Section 4, above, worked very successfully in that all workspaces that were tested worked as expected. Although all of the work spaces have not yet been tested, there is substantial reason to believe that the three man days devoted to moving them is close to the total time required. It was necessary to write APL/VS functions to simulate the input/output functions of APLSF so that workspaces would function as intended. This code is rather more intricate than one would like but it seems to meet the needs. Some workspaces, particularly the verification workspaces, have symbol tables with more than 256 entries. In a default APL/VS workspace, the default symbol table size is 256 entries and these workspaces have more than 256 variables and functions. It was necessary to modify this symbol table size so as to accommodate the larger workspaces.

It has been a consistent policy for this research to avoid the creation of Fortran programs. Nevertheless, a number of Fortran coded library programs are used to support the program verifier. For reasons of efficiency, the parser in the program verifier does not generate parse tables from an input grammer. Rather it releies on an SLR1 parser generator that was coded in RATFOR and translated into Fortran. These programs were represented by their author as being substantially machine independent and easy to transport to other installtions; indeed, they executed correctly using CDC Fortran and both DEC Fortrans. The difficulties associated with transferring these programs has been substantial.

RATFOR is one of a wide variety of syntactical front ends for Fortran. It attempts to provide a reasonable syntax for Fortran programs. The RATFOR compiler, itself a Fortran program, accepts RATFOR statements as input and allegedly writes "standard" Fortran as output. A RATFOR compiler was used to create the Fortran programs that comprise the SLR1 parser generator. The original plan for the transfer of these programs called for transferring the RATFOR compiler as well as the RATFOR source programs for the SLR1 parser generator. As a backup measure, the Fortran versions of the programs in the SLR1 parser generator were also transported; this turned out to be a very wise decision.

The RATFOR compiler is a Fortran program of some 2500 lines. Approximately two man days were devoted to attempting to compile this program using the IBM Fortran compiler. At the end of this time, it was quite obvious that the work remaining to be done was substantial and therefore, attempts to successfully convert the RATFOR compiler were abandoned and attention was directed to directly converting the four Fortran programs that are part of the SLR1 parser generator.

The SLR1 parser generator consists of four programs, each of about 1500 lines. The first of these four programs has been successfully compiled and executed. It is not possible to say whether the program runs correctly because the output of this first program is coded input to the second program. There is reason to believe that it runs correctly but there is no evidence to confirm this. The remaining three programs in the SLR1 parser have not yet been converted. This converesion will be attempted as soon as time is available. The unavailability of these programs does not impair current research work but it will become a problem in the future. It is estimated that one to two man weeks of labor are required to complete this conversion.

It is easiest to describe the Fortran incompatibilities by enumerating language features that are not supported by IBM Fortran but are available in CDC and DEC Fortran. ACCEPT and TYPE statements for terminal (or SYSIN/SYSOUT) input and output are not available in IBM Fortran. Hollerith assignments, e.g., A = 'XY' or A = 2HXY, are not available in IBM Fortran. In addition, quoted Hollerith constants are only accepted in FORMAT statements and are rejected in DATA statements. The domain of the logical and relational operators in IBM Fortran is much more restrictive. IBM Fortran accepts lower case characters only in Hollerith constants while CDC and DEC Fortran accepts lower case program text. Lower case was used in these programs and this change forces additional editing.

The file management support provided by the DEC-10 and IBM operating systems differ sharply and these differences have made programs transfer considerably more difficult. At best, there has been some loss in the interactive properties of programs as a result of this move. This reduction in interactive properties is principally related to the use of data files. The problem arises as follows:

the DEC-10 operating system permits users to create or access a file by simply mentioning the name of the file. Thus, it is possible to prompt the user for the name of a file and then open the file to read or write. In contrast, in the IBM operating system, it is necessary to define all files to be used by a program before program execution begins and once execution has started, it is not possible to change files. (This can be done using assembly language code that violates operating system conventions and will not be used because it is poor programming practice and may lead to programs which do not work correctly in the future.) A set of operating system macros, called EXEC files, are being written to prompt the user for file names and to execute the appropriate FILEDEF commands. While this arrangement preserves some of the interactive properties of the DEC-10 environment, it suffers from the disadvantage that decisions must be made before the requisite information is at hand and this makes using programs more difficult.

In summary, the following has been accomplished:

(1) The SIMULA implementation of APL as described in Section 2, above, has been successfully moved from the DECsystem-10 at the University of Arizona to the IBM 370/158 at Virginia Tech. The program is working correctly and in substantially the same way that it did at the University of Arizona.

(2) The program verification system described in [Britton, 1977] has been moved from the University of Arizona to Virginia Tech. This program appears to be working correctly although all paths through the program have not yet been tested. There is no reason to anticipate further difficulties.

(3) APLSF workspaces used in this research and in other tasks have been transferred from the University of Arizona to Virginia Tech.

(4) The files for the Fortran programs that are used to generate parser tables for the SLR1 parser in the program verifier have been successfully moved. One of these four programs has been compiled and executed; it appears to work correctly but final testing awaits the conversion of remaining programs.

6. NAMESPACES FOR APL

At a workshop sponsored by Syracuse University and STAPL held at that University's Minnowbrook Conference Center in September 1977, a number of sessions were devoted to extensions of APL to provide for the protection of groups of functions and variables, to facilitate library management and to provide co-routines. The ideas presented at that workshop might well be described as preliminary ideas about a new extension to APL.

During these sessions and in conversations with the speakers, it became obvious that this proposed extension to APL closely resembles the class concept of SIMULA. In order to clearly focus on these ideas, an informal discussion of SIMULA classes together with a class-like extension to APL was written. This manuscript was complete early in January, 1978 and was typed with grant support but duplicated by the University of Arizona. This informal discussion has been submitted, as an extended abstract of a paper, for presentation at APL 79 in Rochester, New York in May 1979. A copy of the manuscript is Attachment 5 to this report.

The basic view adopted in this manuscript may be described briefly as follows. For at least four years, there has been an active debate concerning arrays of arrays as an extension to APL in the APL community. This debate has led to considerable new understanding of the properties of arrays of arrays but the work has not proceeded to a point where it is possible to clearly define a reasonable extension of APL to include arrays of arrays in an unambiguous way. This leads me to believe that the concept of arrays of arrays is not an appropriate extension to APL. Yet, one frequently finds that this general kind of extension to APL is useful when coding a particular algorithm or choosing a data representation before coding an algorithm. This suggests that some structure of this sort would be a useful extension. The SIMULA class, with its clean and orderly definition, provides a simple and straightforward way to extend an underlying language to essentially arbitrary data structures. Moreover, these data structures can be defined by the user in such a way that a node of the structure may consist of both storage locations and procedures. Such a structure can be traversed in many different ways and at each node one can either refer to the data stored there or execute procedures that are attributes of a node. This kind of extension to APL will provide a solution for the problems that are being addressed by arrays of arrays.

The great strength of APL lies in its ability to apply a primitive or user defined function to either a scalar or to an array of an arbitrary number of dimensions. A class-like extension of APL should include the possibility of extending primitive functions and user defined functions to data structures constructed with a class mechanism. This can be achieved quite simply by including a defi-

nition of the way in which a monadic function is to be applied to an instance of the class and by defining the way in which a dyadic function is to be applied to an instance of the class. With these two definitions, it is possible to generalize all of the functions to the arbitrary structures that the user defines and, therefore, I have proposed that these definitions be part of a class-like extension to APL.

This work is peripheral to the main thrust of the grant but, nevertheless, is not completely independent of it. Extensions to APL that cause APL's character to be substantially changed would dramatically reduce the practical utility of the work that is to be done under the grant. Secondly, the discussion of such language extensions motivates questions about the details of the construction of this present semantics of APL. For example, if a highly regarded proposed extension were to be found to be incompatible with the semantics, this would be evidence to indicate that the semantics is not well constructed. On the other hand, examples of extensions of APL that can be easily added to the semantics support the claim that the semantics is amenable to generalization and can, to some extent, be used to support the claim that the proposed extension is reasonable. It appears to be very easy to add this class-like extension to the semantics and implementation that are part of the grant work.

7. PERSONNEL AND ACTIVITIES

Dr. Richard J. Orgass devoted approximately five man months to the work reported here. Of these five months, two months were supported by grant funds. In addition, two months of work were done while Dr. Orgass was supported by the University of Arizona and one man month of work was done while Dr. Orgass was supported by Virginia Polytechnic Institute and State University.

During this time, Dr. Orgass was engaged in developing the semantics of APL described in Section 1, above, and supervising the work of students employed by the grant. In addition, he made a number of contributions to the design of the program described in Section 2, above and began construction of a proof of its correctness.

Mr. Ralph B. McLaughlin was supported by grant funds for the total of 4.5 man months. This appointment consisted of five months with a half time appointment and two months with a full time appointment. During this time, Mr. McLaughlin's responsibilities were the design and verification of the program described in Section 2. A fragment of this program was already in existence when the grant began and so the program design consisted primiarly of modification and extension of an existing program. This was a large piece of work but certainly not one that would have occupied all of this time. A review of Mr. McLaughlin's work was conducted at the end of June and it was determined that it was highly unlikely that he could successfully participate in the work of this grant and complete his Ph.D. requirements. Therefore, Mr. McLaughlin's association with the work of this grant was ended in July 1977 and he will not further participate in this research.

Mr. Karl Rautenkranz, a graduate student at the University of Arizona was supported by this grant for four man weeks. Mr. Rautenkranz wrote the APL workspace transfer programs described in Section 4, above, and, in addition, performed a wide variety of editing tasks in support of the transfer of this research from the University of Arizona to Virginia Tech. This proved to be a highly satisfactory relationship for all concerned because Mr. Rautenkranz did high quality work and learned a great deal at the same time.

A student secretary was supported by the grant during the spring semester and a second student secretary was supported by the grant during the summer. The use of student secretaries considerably reduced salary expenses and in the case of the spring term secretary outstanding support was provided by a highly qualified secretary. The student employed during the summer was useful but only marginally qualified for the job and, in balance, the engagement of this person was close to a mistake.

8. PUBLICATIONS

The results of the research on the semantics and implementation are to be published as a research monograph. A partial draft of the monograph together with a detailed summary of the remainder of the monograph will soon be submitted for consideration for the Lecture Notes in Computer Science published by Springer-Verlag and for the Computer Science Series published by Elsiever-North-Holland. In order to make some of this work available before publication, the first two chapters were issued as technical reports:

> R. J. Orgass and R. B. McLaughlin. A Formal Semantics and Implementation of APL, Chapter I - Introduction. Technical Report No. APLTD7, Department of Computer Science, University of Arizona, November 18, 1977.
>
> R. J. Orgass and R. B. McLaughlin. A Formal Semantics and Implementation of APL, Chapter II - Semantics of Simple Expressions. Technical Report No. APLTD8, Department of Computer Science, University of Arizona, May 15, 1978.

These reports are Attachment 1 to the present report and the distribution list is Attachment 2.

A set of DEC-10 programs that were used to transfer files to an IBM System 370 were written while this research was transferred from the University of Arizona to Virginia Tech. These programs are described in:

> R. J. Orgass. Transferring Files to an IBM Machine. Technical Memorandum No. APLAD14, Department of Computer Science, University of Arizona, July 28, 1978.

This work was supported by grant funds and a limited number of copies were duplicated using grant funds; additional copies were duplicated using Virginia Tech funds. This report is Attachment 3 to the present report.

Grant funds were used to write APL programs to transfer APL workspaces from the University of Arizona to Virginia Tech. After the grant terminated, a technical report was written:

> K. Rautenkranz and R. J. Orgass. Transfer of APL Workspaces: A Useful Solution. Technical Report No. CS78006-T, Department of Computer Science, Virginia Polytechnic Institute and State University, September 1978.

This report, which is Attachment 4 to the present report, was prepared and duplicated with Virginia Tech funds.

Before grant support began, some work on a namespace extension to APL was completed. This work is described in:

R. J. Orgass. Concerning Namespaces for APL. Technical Memorandum No. APLAD12, Department of Computer Science, University of Arizona, February 10, 1978.

This report, which is Attachment 5 to the present report, was prepared using grant funds and duplicated with funds provided by the University of Arizona and Virginia Tech. This report was submitted for presentation at APL79 and has been accepted as an extended abstract. A review of the final manuscript is pending.

Work on program verification that is the starting point for research under this grant was completed before the grant began. This work is described in:

D. E. Britton. Incremental Synthesis of Inductive Assertions. Ph.D. Dissertation, University of Arizona, 1977.

During the term of Grant AFOSR-79-0021, the third chapter of the research monograph will be issued as a technical report. The remaining chapters will first appear in the monograph.

9. REFERENCES

[Britton, 1977]
D. E. Britton. Incremental Synthesis of Inductive Assertions for Program Verification. Ph.D. Dissertation, University of Arizona, 1977.

[Feldbrugge, 1973a]
F. H. J. Feldbrugge Een Qpzet van eeen Interaktief Systeem voor de Verifikatie van APL-programma's (ISVAP). Ph.D. Dissertation, Technische Hogeschool Twenk, 1973.

[Feldbrugge, 1973b]
F. H. J. Feldbrugge. An Interactive System (ISUAP) for the Verification of APL Programs Using Floyd's Method in P. Gjerløv, H. J. Helms and J. Nielsen (Eds.) APL Congress 73. Amsterdam, North-Holland, 1973, pp. 119-126.

10. LIST OF ATTACHMENTS

The following attachments are provided only with the original of this report.

Attachment 1
Technical Reports on the Semantics of APL

R. J. Orgass and R. B. McLaughlin. A Formal Semantics and Implementation of APL, Chapter I - Introduction. Technical Report No. APLTD7, Department of Computer Science, University of Arizona, November 18, 1977.

R. J. Orgass and R. B. McLaughlin. A Formal Semantics and Implementation of APL, Chapter II - Semantics of Simple Expressions. Technical Report No. APLTD8, Department of Computer Science, University of Arizona, May 15, 1978.

Attachment 2
Distribution List for Attachment 1

Attachment 3
Technical Memorandum on File Transfer

R. J. Orgass. Transferring Files to an IBM Machine. Technical Memorandum No. APLAD14, Department of Computer Science, University of Arizona, July 28, 1978.

Attachment 4
Technical Report on APL Workspace Transfer

K. Rautenkranz and R. J. Orgass. Transfer of APL Workspaces: A Useful Solution. Technical Report No. CS-78006-T, Department of Computer Science, Virginia Polytechnic Institute and State University, September 1978.

Attachment 5
Technical Memorandum on Namespaces

R. J. Orgass. Concerning Namespaces for APL. Technical Memorandum No. APLAD12, Department of Computer Science, University of Arizona, February 10, 1978.

APPENDIX I

Partial Manuscript
of

A PRIMITIVE RECURSIVE SEMANTICS AND IMPLEMENTATION OF APL

CHAPTER III

Implementation of Simple Expressions

CHAPTER III

IMPLEMENTATION OF SIMPLE EXPRESSIONS

Chapter III Appendicies

E. INTRODUCTION TO SIMULA

F. FUNCTIONS AND PREDICATES ON INDIVIDUALS

G. IMPLEMENTATION OF MONADIC SCALAR FUNCTIONS

H. IMPLEMENTATION OF DYADIC SCALAR FUNCTIONS

I. IMPLEMENTATION OF MIXED FUNCTIONS

CHAPTER III
IMPLEMENTATION OF SIMPLE EXPRESSIONS

30. Overview

An implementation of the simple expression evaluator apl is described in this Chapter and it is shown that the program evaluates expressions in accord with the definitions given in Chapter II. With minor exceptions that are pointed out in the text, the program described here is part of the complete APL interpreter described in Chapter VIII and the proof of correctness of this program is part of the proof of correctness of the complete interpreter.

There are three major reasons for constructing this implementation and proving that it is correct.

I wish to claim that primitive recursive arithmetic, as used in Chapter II, is a suitable tool for defining real programming languages, for constructing implementations of programming languages and for proving the correctness of an implementation. The construction and verification of the present implementation of APL provides evidence to support this claim with respect to one widely used interpretive language.

The formal definition of a widely used programming language is a difficult task because it is necessary to provide assurances that the formal definition matches the language as it is used in practice. A substantial number of examples have been executed on one or more implementations of APL in order to determine the precise properties of primitives and the results of these examples have been used to construct the definition of the syntax and semantics of simple expressions. Nevertheless, there are many more complicated examples that should be executed with the semantics to confirm that the language defined in the semantics matches the language as it is used. Since it is shown that the implementation is correct with respect to the semantics, the program can be used to execute these examples much more quickly and reliably.

In another place, the author will show that the formal semantics given here can be used to prove the correctness of APL programs. A persistent question in program verification is: "Suppose a program has been shown to be correct using some formal semantics of a programming language, how do we

know that the program will run correctly? The implementor of the language may have made mistakes in the implementation." In this case, the question does not arise because it has been shown that the implementation is correct with respect to the formal semantics that is used to verify a program.

The choice of a programming language for the implementation was fairly easy after I wrote down my specifications for the language. The following requirements guided my choice: (1) The language must be well defined so that it is easy to prove statements about programs written in the language. (2) The language must be widely used and available on a number of different computing machines with careful control of implementations. (3) The language must have a powerful abstraction mechanism so that the implementation can easily be structured. (4) The language must provide security mechanisms so that parts of the implementation can be protected from each other. (5) The object code written by the compiler must be reasonably efficient.

There are four languages known to me that meet some of these requirements: ALPHARD, CLU, PASCAL and SIMULA-67. ALPHARD and CLU were rejected because the definition of these languages is changing and because they are not available on a variety of machines. They are not widely used for practical programming and there is some evidence to suggest that the object code written by these compilers is quite inefficient. PASCAL meets many of my requirements but the abstraction mechanism of PASCAL, essentially record types, is inadequate for this work. This statement can be confirmed by considering the program described in this Chapter. There are essentially no protection mechanisms beyond checking a variable for values in its range and this significantly complicates the proof of correctness of the program.

SIMULA-67 meets the above requirements quite well. The common base definition [1] is, essentially, an extension of the Algol-60 report [2] and it is possible to use most of the existing work on the verification of Algol programs, e.g. [3,4], in this work. SIMULA has been implemented on a variety of computing machines and these implementaions are carefully controlled by the Norwegian Computer Center. The class concept of SIMULA provides a powerful abstraction mechanism (for example, see [5]) and when this is combined with the protection mechanisms of the language it also provides excellent security that is enforced at compilation.

It is assumed that the reader is familiar with SIMULA. In order to make this volume self-contained, a brief informal introduction to SIMULA

appears in Appendix E. The reader is referred to Dahl's paper [6] for a more extensive discussion and to the common base definition [1] for a precise definition of the language.

When designing this implementation, the objective was to construct a reasonably efficient program that is easy to verify. There are a number of optimizing and correctness preserving transformations that can be applied to the program. If they are applied, the resulting program would be less obvious and many of the proofs would be more complicated and, therefore, the program has not been optimized in great detail. In many circumstances, it was a pleasant surprise to find that a program structure that facilitates verification is also more efficient than alternative structures.

The organization of this Chapter closely follows the organization of Chapter II. The text describes the essential arguments and routine details appear in appendicies.

Section 31 describes the implementation of APL individuals and it is shown that the implementation is equivalent to the definition given in Section 21.4. The implementation of primitive scalar functions is described in Section 32 and it is shown that the programs that implement these functions are correct with respect to the semantics. Section 33 has a similar discussion for mixed functions.

The implementation of symbol tables is described in Section 34 and it is shown that the implemented symbol tables are equivalent to the symbol tables defined in Section 26.2. Symbol tables are used by both the parser and the expression evaluator. The syntax analyzer is described in Section 35 and it is shown that the set of expressions accepted by the syntax analyzer is the set of expressions defined in Sections 24 and 27.3. In addition to recognizing expressions, the syntax analyzer constructs an expression tree that is used by the expression evaluator. It is shown that the trees that are constructed have certain properties that are used to prove that the expression evaluator is correct.

The discussion of function application and assignment is part of the description and verification of the expression evaluator. Section 36 describes the function application mechanism and this is shown to be equivalent to the definitions of Section 26.3. Section 37 describes the implementation of assignment and this implementation is shown to be equivalent to the definitions of Sections 27.1 and 27.2.

Finally, in Section 38, the implementation of the simple expression evaluator apl is described and it is shown that the value of an expression, as computed by the interpreter is the same as the value of the function apl of Section 27.4 for all simple expressions and symbol tables.

31. APL Individuals

The most difficult task in the construction of an APL interpreter based on the present semantics is the design of the data structure used to implement APL individuals and the verification of this design with respect to the formal semantics. The initial design was modified many times in order to make later parts of the implementation simpler, more efficient and easier to verify. A final set of modifications was made to make it possible to prove the theorems about the implementation that appear in Section 31.2.9, below. It was a pleasant surprise to find that these last modifications further simplified later parts of the implementation.

The discussion of the implementation of APL individuals is primarily directed toward describing the data structure and to proving that it satisfies the definition of APL individuals given in Section 21.4 but the presentation includes some of the problems that were encountered during the design; many additional design problems are not discussed at all.

Section 31.1 discusses the implementation of APL numbers and characters and closely follows Section 21.2. It is important to note that the internal representation of APL characters is independent of the character code of the host machine. Translations from and into the character set of the host machine occur immediately after input is read and just before output is written. This makes it possible to move the present implementation from one machine to another with minimal changes and to support different terminals easily. In fact, this implementation was moved from a DECsystem-10 to an IBM System 370 while work on this Chapter was in progress.

The main part of this Section is the verification of the implementation of APL individuals in 31.2. The discussion begins with simple examples to introduce the major conceopts and ends with a proof of the correctness of the implementation.

31.3 Numbers and Characters

APL numbers, booleans and characters are implemented using the SIMULA type real, that is, floating point numbers. Booleans are implemented as floating point 0 and 1 and rationals are implemented using floating point numbers that approximate these numbers. Characters are mapped into floating point numbers using a function that is similar to the pairing function used in Section 21.2.

In the formal semantics, arithmetic on rationals is infinite precision. In many circumstances, this arithmetic is adequately approximated by single precision floating point arithmetic but in some cases this approximation is inadequate and it is necessary to use extended precision arithmetic. Since we do not wish to address the numerical analysis associated with a choice of word length, we have assumed that the single precision arithmetic of the host machine, as used by the SIMULA compiler, is an adequate approximation of the arithmetic on rational of the formal semantics. If this approximation is inadequate, double precision arithmetic may be used by replacing all occurrances of the keyword REAL by the keywords LONG REAL in the text of the program.

It is possible to implement arbitrary precision arithmetic using an alternate representation of rationals and procedures for the arithmetic operators. This approach was rejected because it is extremely inefficient and because it does not conform to the usual implementations of APL.

Each APL character consists of two simple characters. Let *kp1* and *kp2* be the character codes for the two simple characters that form an APL character as given in Appendix A. This APL character is mapped into a floating point number using a function that is described by the expression:

```
if kp1 ≤ kp2
   then kp1*128 + kp2
   else kp2*128 + kp1
```

The two obvious functions can be used to extract the character codes of the two simple characters that form the APL character.

If the mantissa of floating point numbers has at least 14 bits, it is easy to see that this function is a 1-1 into map from APL characters to floating point numbers with a left and right inverse. Further, the floating point relations less than and greater than, respectively, are equivalent to the relations cless and cgreater on APL characters for this implementation of APL

characters (see Sections 21.3 and B3).

In order to simplify the following discussion, the SIMULA keyword real will be used to refer to floating point numbers.

The function that maps APL characters into reals is implemented by the function *encode* shown in Figure 31.1a. The return value of the function *kp_code* is the number of its simple character argument as given in Appendix A. The exact declaration of *kp_code* depends on the character set of the host computing machine and the APL terminal that is used. For each such case, it must be shown that this function has the above property.

```
REAL PROCEDURE encode(c1,c2); CHARACTER c1, c2;
BEGIN
    REAL kp1, kp2;
    kp1 := kp_code(c1);
    kp2 := kp_code(c2);
    encode := IF kp1 <= kp2
                  THEN kp1 * 128 + kp2
                  ELSE kp2 * 128 + kp1
END of encode;
```

(a)

```
TEXT PROCEDURE decode(n); REAL n;
BEGIN
    TEXT temp;
    IF n <= 127
       THEN BEGIN
                 temp :- Blanks(1);
                 temp.putchar(kp_char(n));
                 decode :- temp
            END
       ELSE BEGIN
                 temp :- Blanks(3);
                 temp.puchar(kp_char(n // 128));
                 temp.putchar(kp_char(back_space));
                 temp.putchar(kp_char(Mod(n,128)));
                 decode :- temp.strip
            END
END of decode;
```

(b)

Figure 31.1

The return value of the function *kp_char* is the character string (or text) that is to be sent to an APL terminal to print the simple character that corresponds to it argument character number as given in Appendix A. The definition of this function also depends on the character set of the host computing machine and the APL terminal that is used. The program described here has been executed on a DECsystem-10 and on an IBM System 370; the only change made in the program was the definition of *kp_code* and *kp_char*.

An inverse of *encode* that maps APL characters into their printed form is used for output. The function *decode*, shown in Figure 31.1b, implements this mapping. The parameter of this function is an APL character and the return value is a text object which, when transmitted to a terminal or file, will print the APL character. If the APL character is less than or equal to 127, then the character consists of a single printable character because the second character in such pairs is character number 0. On the other hand, if the APL character consists of two characters, then the printed form of the character consists of three characters: the first character, a backspace and the second character. (The value of the variable *back_space* is the character number of the backspace character as given in Appendix A.) These statements plus the definition of the SIMULA primitives putchar, strip, // and Mod can be used to show that *decode* has the claimed property.

31.2 Implementation of Individuals

In the formal semantics, an APL individual is a single object with five attributes and each of these attributes themselves have additional properties. The set of APL individuals is implemented as a SIMULA class, called the class of individuals, and each instance of this class is an individual whose attributes have the same properties as the attributes of APL individuals in the formal semantics. It is shown that each member of a large finite set of APL individuals is implemented by an instance of the class of individuals and that each instance of the class of individuals has all of the properties of an ALP individual in the formal semantics.

In this chapter, it is frequently necessary to use identifiers from the implementation in the text. Thses identifiers appear in *light italic* so they can be recognized. Examples of program text appear in Algol publication format and actual program text, which was printed on a terminal, is exhibited using upper case keywords and lower case for the remaining identifiers.

This section begins with a simple example of a SIMULA implementation of an APL individual (31.2.1) and this example motivates the definition of a computer representation implementing an APL individual (31.2.2). This definition is used to show that each APL individual whose attributes can be represented in a machine is implemented by an instance of the simple class of individuals.

The simple declaration of individuals is inadequate because it is not possible to show that each instance of the class has all of the properties of an APL individual. A restricted individual that solves some of the problems is described (31.2.3) and this declaration motiviates the more extensive declaration that is discussed in Sections 31.2.4 to 31.2.9. It is shown that each instance of the class of individuals in the implementation has all of the properties of an APL individual and that each APL individual whose attributes can be represented in the host computing machine is implemented by an instance of this class.

The Section concludes with a discussion of some implementation issues which greatly simplify the final program.

31.2.1 Example

It is easiest to begin by examining a substantially simplified declaration of class *individual*:

```
class individual(rank, cardinality);
  value rank, cardinality; integer rank, cardinality;
begin
  integer type;
  integer array shape[1:rank];
  real array elements[1:cardinality];
end of individual;
```

When an instance of this class is created, the *rank* and *cardinality* of the new *individual* are passed as parameters so that the run time system can allocate storage for the arrays *shape* and *elements*.

Each instance of this class is a single object and may be the value of a SIMULA variable [of type ref(*individual*)]. The integer values of the variables *rank*, *cardinality* ($|X|$) and *type* correspond to the attributes of APL individuals with the same names. In the formal semantics, the shape attribute of an APL individual X is a rank(X)-tuple of natural numbers and the array

shape corresponds to this attribute. Similarly, the elements attribute of an APL individual X is a $|X|$-tuple of elements. APL numbers, booleans and characters are all implemented as reals and, thus, the array *elements* corresponds to the elements attribute of an APL individual.

In summary, if b is an instance of class *individual* (b is an *individual*) then the correspondence between the attributes of b and the attributes of an APL individual B in the formal semantics is as follows:

Class Individual	APL Individual
b.rank	rank(B)
b.shape	shape(B)
b.cardinality	$\|B\|$
b.type	type(B)
b.elements	$\{B\}$
b.elements[*i*]	$\{B\}_i$

There is an obvious correspondence between one dimensional arrays in SIMULA and ordered n-tuples and this correspondence will be used without further comment.

This example provides an informal introduction to the more precise treatment of the implementation of APL individuals that follows.

31.2.2 Implementation

Definition. A computer representation of an APL individual is said to implement this individual if it is a single object with the five attributes of an APL individual and if:

(1) The rank, cardinality and type attributes of the computer representation, when interpreted in accord with the representation of integers in the host machine, are the same as the corresponding attributes of the APL individual.

(2) The shape attribute of the computer representation has the same number of components as the shape attribute of the APL individual and each component of the computer representation, when interpreted in accord with the representation of integers in the host machine, is the same as the corresponding component of the shape attribute of the APL individual.

(3) The elements attribute of the computer representation has the same number of components as the elements attribute of the APL individual and if one of the following relations holds, component for component, between the computer representation and the APL individual:

(a) If the APL individual is of type boolean, then the component of the representation is interpreted as a 0 or 1, in accord with the value of the component of the APL individual.

(b) If the APL individual is of type character, then the component of the representation, when interpreted for printing on an output device, is the APL character in the APL individual.

(c) If the APL individual is of type number, then the component of the realization is the floating point number of the host machine that is the closest approximation of the rational in the APL individual.

A set of APL individuals is said to be implemented by a set of computer representations if each APL individual is implemented by a member of the set of representations and if each representation in the set implements an APL individual.

This definition will be used to prove statements about a SIMULA implementation of APL but it could equally well be used to prove statements about implementations using languages such as PASCAL, PL/I, etc.

It is straightforward to verify that the representation of APL numbers, booleans and characters described in Section 31.1 meets the requirements of (3a) to (3c).

Each SIMULA implementation provides the function procedures *maxint* and *maxreal* whose return values are the largest integer and the largest floating point number, respectively, that can be stored in a word of the host machine; these identifiers are used below. The discussion assumes that sufficient machine resources, e.g., storage, are available.

Lemma 1. Each APL individual whose rank and cardinality are less than or equal to *maxint* and such that each component of its shape is less than or equal to *maxint*, is implemented by a member of *individual*.

Proof. The lemma is established by exhibiting an algorithm for creating the appropriate member of *individual* given an APL individual that satisfies the conditions stated in the lemma. It is assumed that the integers one to

thirteen (inclusive) can be stored in a single word of the host machine so that a type number can be stored in a single word.

Step 1. Let r and c, respectively, be computer integers that correspond to the values rank(X) and $|X|$. By the hypothesis of the lemma, $r \leq maxint$ and $c \leq maxint$. Create an instance of *individual* by executing the statement:

x :- new *individual*(r,c);

By the definition of new and of value parameter transmission, $x.rank = r$ and $x.cardinality = c$. Therefore, *rank* and *cardinality* attributes of x satisfy (1) of the definition.

Step 2. Let $s_1, s_2, \ldots, s_r$ be computer integers that correspond to the components of shape(X). Set the values of the r components of $x.shape$ using assignments of the form:

$x.shape[1] := s_1;$
$x.shape[2] := s_2;$
...
$x.shape[r] := s_r;$

After these assignments are executed, the *shape* attribute of x satisfies (2) of the definition. If r is zero, then *shape* is not initialized.

Step 3. Let t be the computer integer that corresponds to type(X). Set the *type* attribute of x with the assignment:

$x.type := t;$

After this assignment is executed, the *type* attribute of x satisfies (1) of the definition.

Step 4. Let $u_1, u_2, \ldots, u_c$ be computer representations of the components of $\{X\}$ as described in Section 31.1. Set the values of $x.elements$ using assignments of the form:

$x.element[1] := u_1;$
$x.element[2] := u_2;$
...
$x.element[c] := u_c;$

After thses assignments are executed, $x.elements$ satisfies (3) of the definition because the representations of Section 31.1 satisfy (3a) to (3c).

This completes the proof.

31.2.3 Restricted Individuals

Unfortunately, this simple declaration of *individual* is inadequate because it is not possible to prove that each *individual* implements an APL individual. The proof fails because there are members of *individual* that do not implement individuals. For example, it is possible to set *cardinality* to a value that is not the product of the components of *shape*. Similarly, it is possible to set *type* to 3 when the components of *elements* do not correspond to characters under the mapping described in Section 31.1. Moreover, even if an *individual* is correctly initialized, the attributes may be accidentially changed by the program that manipulates *individuals* so that it no longer has the properties of an APL individual.

It is, in principle, possible to prove that the program which contains this declaration uses it correctly but this is a long, difficult and error prone process. It is much simpler to use the protection mechanisms of SIMULA to make sure that each *individual* implements an APL individual. The declaration of *individual* discussed in Section 31.2.3 is a first approximation to the declaration that is used in the present implementation.

The declaration of *individual* shown in Figure 31.2 has many of the properties of the declaration used in the present implementation. It is presented to make it easier to understand the actual declaration that is described in Sections 31.2.4 to 31.2.9.

By the definition of protected attributes in SIMULA, each instance of this class, when viewed from outside the class instance, behaves as though its only attributes are *rank*, *shape*, *cardinality*, *type*, and *element*. The remaining attributes of the class instance cannot be referenced outside an instance of this class. It will be shown that these attributes have the properties of the corresponding attributes of an APL individual.

When a member of this class is created, three parameters are passed: the rank of the new individual (*rnk*), the shape of the new individual (*shpe*) and the type of the new individual (*typ*). These parameters are transmitted by value so the copy inside the *individual* is different from the external copy. These attributes are protected so they cannot be changed after the *individual* is created. The cardinality of the new *individual* is not a parameter so that it can be correctly computed when the *individual* is created.

The next step in the creation of an *individual* is storage allocation for the integer *card* (which will contain the *cardinality*) and the *elements*

```
class individual(rnk, shpe, typ); value rnk, shpe, typ;
   integer rnk, typ; integer array shpe;
   not protected rank,
                 shape,
                 cardinality,
                 type,
                 element;
begin
   integer card;
   real array elements[1:compute_card];

   integer procedure rank;
      rank := rnk;

   integer procedure shape(i); value i; integer i;
      shape := shpe[i];

   integer procedure cardinality;
      cardinality := card;

   integer procedure type;
      type := typ;

   real procedure element(i); value i; integer i;
      element := elements[i];

   integer procedure compute_card;
   begin
      integer j;
      card := 1;
      for j := 1 step 1 until rnk do
         card := card*shpe[j];
      compute_card := card;
   end of compute_card;
   if 0 < typ or typ > 13
      then Abort("INDIVIDUAL: Invalid type.");
   if 5 ≤ typ and typ ≤ 11 and (rnk ≠ 1 or shpe[1] ≠ 0)
      then Abort("INDIVIDUAL: Invalid Special Individual.")
      else card := 0;
   if rnk = 0 and shpe[1] ≠ 0
      then Abort("INDIVIDUAL: Invalid scalar.")
end of individual;
```

Figure 31.2

array. The function *compute_card* is called to compute the upper bound of the array. It is straightforward to verify that the following things happen when the function *compute_card* is called. If the value of *rnk* is zero, then *card* is set to one; otherwise it is set to the product of the first *rnk* components of *shpe; card* is also the return value of the procedure. Two errors may occur at this point: If *shpe* has more than *rnk* components, all components of *shpe* after *shpe*[*rnk*] will be ignored in the computation of *card* so the instance satisfies the definition of an APL Individual. On the other hand, if *shpe* has less than *rnk* components, then execution will be terminated as a result of an array bound error. Therefore, if the program is in execution after storage allocation is complete, the individual has the correct relationship between its rank, shape and cardinality. [see statements (3) and (4) in the definition of APL individuals, Section 21.4.]

The last step in the creation of an *individual* is the execution of the statements in the main block. Each of these statements will terminate execution using the procedure *Abort* if the *individual* does not satisfy the definition of an APL individual. The first statement checks to see that the type is acceptable [21.4, clauses (1) and (2)]. The second statement checks to see that the special individuals have the proper type [definition of special and destinguished individuals, and clause (1)] and sets the cardinality of such individuals to 0. The third clause checks to see that labels have the proper attributes [clause (10) of 21.4] and the fourth clause checks to see that scalars have the appropriate attributes [clause (3)]. The components of *elements* satisfy the requirements for components of the elements attribute because these components are initialized to zero and this is an acceptable boolean, number or character. Therefore, if an *individual* is created and execution continues to the end of the creation process, this new *individual* has the properties of an APL individual because it satisfies the definition given in 21.4.

It is straightforward to confirm that the procedures *rank, type* and *cardinality* have the corresponding attributes of an APL individual as their return value. The attributes *shape* and *element* provide access to the components of the corresponding attributes of the *individual*. The storage locations that contain the values of the attributes of the individual are protected and, therefore, they cannot be changed by any program which manipulates individuals. Since each *individual* is created with the correct values

for its attributes and these attributes cannot be changed, it has been shown that:

Lemma 2. As long as execution continues, each instance of class *ndividual*, as declared in Figure 31. has the attributes of an APL individual.

It is straightforward to modify steps 1, 2 and 3 of the algorithm in the proof of Lemma 1 for use with this declaration of *individual*. However, step 4 cannot be modified in a satisfactory way because it is impossible to change the values of components of *elements* after an *individual* is created. This problem can be avoided by adding another (not protected) procedure attribute, *set_element*, to the declaration of *individual*. Here is an example of such a declaration:

```
procedure set_element(i,v); value i, v;
  integer i; real v;
if (typ > 3 and typ < 13)
  then Abort("INDIVIDUAL: Element assignment prohibited.")
  else if typ = 1 or
    (typ = 2 and (v = 0 or v = 1)) or
    (typ = 3 and (0 ≤ v and v ≤ 16383))
  then elements[i] := v
  else Abort("INDIVIDUAL: Element out of range writing.");
```

If the value that is to be assigned to a component of *elements* does not meet the requirements for a component of the *individual*, then execution is terminated. Similarly, if the component number is not a correct subscript for *elements*, execution is terminated because a subscript is out of range.

These remarks are an outline of a proof of the statement that *individuals* with this procedure attribute remain *individuals* as long as execution continues.

Although the declaration of *individual* in Figure 31.2 is adequate, it has not been adopted for a number of reasons. First, there are far too many errors that result in termination of execution, an unacceptable attribute of any implementation. It is far preferable to use a declaration with the property that the compiler will detect all errors so there will not be any run time terminations. The declaration that is used in the implementation almost meets this requirement. Second, this declaration provides an extremely ineffient implementation of APL individuals. The overhead associated with the complicated checks in *set_element* and the checks when an *individual* is created is substantial.

The ideas used in this declaration play an important part in the declaration of *individual* as it is used in the present implementation. The running version of the declaration is discussed in Sections 31.2.4 to 31.2.9.

31.2.4 Class Individual

The structure of class *individual*, as used in the implementation, is shown in Figure 31.3. Each node of the tree corresponds to a class and the decendants of a node correspond to subclasses of the parent node. This structure was selected to reflect the various parts of the definition of the set of individuals, to provide adequate security and to simplify the coding and verification of the remainder of the implementation.

APL individuals of type number (1), boolean (2) and character (3) are usually manipulated by APL programs and individuals of these types are grouped together as subclasses of *individual* class *usual*. Since it is often necessary to inquire if an individual is of type number or boolean, these two classes are grouped together as subclasses of class *numerical*.

Except for their type, the special individuals have the same attributes so they are all subclasses of *individual* class *special*. The *special individuals* are further divided into the class of *distinguished individuals* and the *e_vector*.

APL individuals of type label and function definition have restrictions on their attributes that are quite different from other APL individuals and, therefore, they are seperate subclasses of class *individual*.

The declaration of class *individual*, shown in Figure 31.4, defines the external attributes of *individuals*. The protected declaration limits the attributes that are accessible outside *individuals* to the five attributes of APL individuals plus the procedure *set_element* which is used to change the value of components of the (inaccessable) *elements* attribute. Each of these procedure attributes have different declarations in different subclasses of *individual* so they are declared as virtual procedures. By the definition of virtual, the procedure that will be used with a particular *individual* is the lowest one in the tree of Figure 31.3.

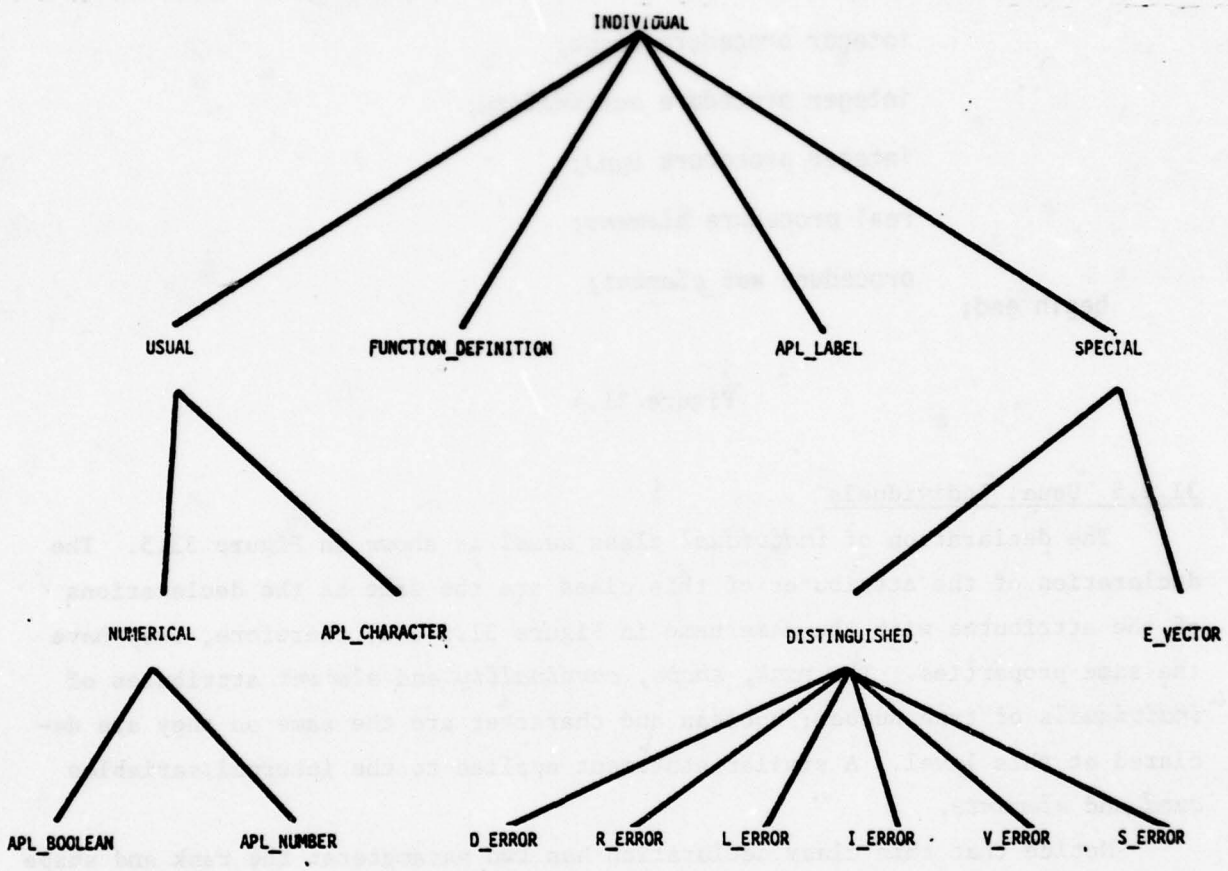

INDIVIDUAL
USUAL
FUNCTION_DEFINITION
APL_LABEL
SPECIAL
NUMERICAL
APL_CHARACTER
DISTINGUISHED
E_VECTOR
APL_BOOLEAN
APL_NUMBER
D_ERROR
R_ERROR
L_ERROR
I_ERROR
V_ERROR
S_ERROR

Figure 31.3

```
class individual;
   not protected rank,
                 shape,
                 cardinality,
                 type,
                 element,
                 set_element;

   virtual : integer procedure rank;

             integer procedure shape;

             integer procedure cardinality;

             integer procedure type;

             real procedure element;

             procedure set_element;
begin end;
```

Figure 31.4

31.2.5 Usual Individuals

The declaration of *individual* class *usual* is shown in Figure 31.5. The declaration of the attributes of this class are the same as the declarations of the attributes with the same name in Figure 31.3 and, therefore, they have the same properties. The *rank*, *shape*, *cardinality* and *element* attributes of *individuals* of type number, boolean and character are the same so they are declared at this level. A similar statement applies to the internal variables *card* and *elements*.

Notice that this class declaration has two parameters: the rank and shape of the individual to be created. The parameter for the type used in Figure 31.2 is not needed.

The declarations of classes *apl_number* and *apl_boolean* are shown in Figure 31.6. Class *numerical* simply serves to group classes *apl_boolean* and *apl_number* into a single class and, therefore, its declaration consists of the empty block.

The *type* attribute of an *apl_number* is simply an integer procedure that returns the value 1. Any real may be a component of the *elements* attribute of an *apl_number* and, therefore, the procedure *set_element* simply changes the value of the appropriate component. If the first actual parameter of

```
individual class usual(rnk, shpe); value rnk, shpe;
   integer rnk; integer array shpe;

begin
   integer card;
   real array elements[1:compute_card];

   integer procedure rank;
      rank :- rnk;

   integer procedure shape(i); value i; integer i;
      shape := shpe[i];

   integer procedure cardinality;
      cardinality := card;

   real procedure element(i); value i; integer i;
      element := elements[i];

   integer procedure compute_card;
   begin comment As in Figure 31.2; end;

   if rnk = 0 and shpe[1] ≠ 0
      then Abort("INDIVIDUAL: Invalid scalar.")

end of usual;
```

Figure 31.5

set_element is not a legitimate subscript for the array *elements*, then the run time system will terminate execution with an error message.

The *type* attribute of an *apl_boolean* is simply an integer procedure that returns the value 2. The components of the *elements* array of an *apl_boolean* must be either 0 or 1 and the procedure *set_element* will terminate execution if an attempt is made to set an element to another value. As is for *apl_numbers*, execution is terminated if a subscript is out of range.

The declaration of *usual* class *apl_character* is shown in Figure 31.7. The type attribute of an *apl_character* is simply an integer proceudre that returns the value 3. The proceudre *set_element* checks to confirm that the new value of a component of the array *elements* is a real that corresponds to an APL character under the mapping described in Section 31.1.

In all of these classes, the only way that a component of the array *elements* can be changed is by means of the procedure *set_element*. This procedure will terminate execution if a proposed new value violates the definition

```
usual class numerical;
begin  end;

numerical class apl_number;
begin

   integer procedure type;
      type := 1;

   procedure set_element(i,v); value i,v;
         integer i; real v;
      elements[i]:=  v;

end of apl_number;

numerical class apl_boolean;
begin

   integer procedure type;
      type := 2;

   procedure set_element(i,v); value i,v;
      integer i; real v;
   if v = 0 or v = 1
      then elements[i] := v
      else Abort("INDIVIDUAL: Boolean out of range.");

end of apl_boolean;
```

Figure 31.6

```
usual class apl_character;
begin

   integer procedure type;
      type := 3;

   procedure set_element(i,v); value i,v;
      if 0 ≤ v and v ≤ 16383 and (v//128) ≤ mod(v,128)
         then element[i] := v
         else Abort("INDIVIDUAL: Character out of range.");

end of apl_character;
```

Figure 31.7

of an APL individual.

It is straightforward, but tedious, to verify the above statements in detail so the more detailed arguments have been omitted.

Lemma 3. Each individual of type 1, 2 or 3 whose rank and cardinality is less than or equal to *maxint* and such that each component of its shape is less than or equal to *maxint*, is implemented by a member of class *apl_number*, *apl_boolean* or *apl_character*, respectively.

Proof. The proof of this lemma is very similar to the proof of Lemma 1. A proof for an APL individual of type 1 (number) is given; the other two cases are essentially the same.

Step 1. Let r be a computer integer that corresponds to the value of rank(X) and let s be an integer array with r components that contains the components of shape(X). Create an *individual* by executing the statement:

```
x :- new apl_number(r,s);
```

By the definition of new and of value parameter transmission, the variables *rnk* and *shpe* will be copies of r and s. It is straightforward to verify that these attributes are available outside x using the procedures *rank* and *shape*. It has already been shown that the value of *card* is computed correctly and that *elements* is created with the correct number of components. This means that the proceudres *cardinality* and *element* correctly transmit the values of the corresponding attributes. The procedure *type* always returns the value 1, as required.

Step 2. Let e be an array of *cardinality* components such that $e[i]$ is the value of the ith component of the elements attribute of the APL individual. Execute the following statement:

```
for i := 1 step 1 until cardinality do
  set_element(i,e[i]);
```

It has already been shown that the result of executing *set_element(i,v)* is to change the ith component of *elements* to v provided that v is an acceptable value. Since each APL individual meets the requirements of the definition given in Section 21.4, all assignments will be done without error.

By definition of implementaion, this new *individual* implements the given APL individual.

Lemma 4. As long as any program containing the declaration of *individual* in Figures 31.4 to 31.7 is running without error termination, each member of *apl_number*, *apl_boolean* and *apl_character* implements an APL individual.

Proof. It must be shown that each instance of these classes satisfies clauses (2) to (7) of the definition of APL individuals given in Section 21.4.

(2) The declarations of the procedures *type* are such that only 1, 2 or 3 are returned.

(3) It has been shown that if *rnk* is 0 then *card* is set to 1. These values are transmitted by the procedures *rank* and *cardinality*. If the first (and presumably only) component of *shpe* is not 0 when *rnk* is 0, the statement in *usual* will terminate execution.

(4) It has been shown that the value of *card* is set to the value specified and this value is transmitted by the procedure *cardinality*.

(5) By the definition of implementation, a member of the set of rationals is implemented by a real. All components of *elements* in *apl_number* must be reals as a result of SIMULA variable type checking. These are the only *individuals* of *type* 1.

(6) The only *individuals* of *type* 2 are *apl_booleans*. The components of the *elements* array are initialized to 0 and the procedure *set_element* of *apl_booleans* permits only the assignment of the values 0 or 1 to components of *elements*. Therefore, all instances of *apl_boolean* and all *individuals* of *type* 2 satisfy this requirement.

(7) The only *individuals* of *type* 2 are *apl_characters*. The components of the array *elements* are initialized to 0, a number that corresponds to a character under the mapping of Section 31.1. The procedure *set_element* of *apl_characters* permits only the assignment of reals that correspond to characters under the mapping of Section 31.1. Therefore, all instances of *apl_character* satisfy this requirement.

This completes the proof.

31.2.6 Special Individuals

The declaration of *individual* class *special* is shown in Figure 31.8. Since all of the special APL individuals have the same rank, shape, cardinality and elements attributes, these attributes are declared in class *special*. Observe that all of these attributes are included in the declaration so it is impossible to create an instance of a special class with the wrong value of one of these attributes. A compile error will occur if parameters are passed to these classes when an instance is created.

```
individual class special;
begin

   integer procedure rank;
      rank := 1;

   integer procedure shape(i); value i; integer i;
      shape := 0;

   integer procedure cardinality;
      cardinality := 0;

   real procedure element(i); value i; integer i;
      element := 0;

end of special;

special class e_vector;
begin

   integer procedure type;
      type := 5;

end of e_vector;
```

Figure 31.8

The special classes differ only in their type attribute. The declaration of *special* class *e_vector* also appears in Figure 31.8. This declaration contains only an integer procedure that returns the type number, 5 for an *e_vector*.

The *special* class *distinguished*, declared in Figure 31.9, serves only to group the *distinguished individuals* into a single class and, therefore, its declaration consists of an empty block. Each of the *distinguished individuals* is a subclass of this class and consists of a procedure *type* that returns the appropriate type number.

Lemma 5. Each special APL individual is implemented by an instance of the class *individual* as declared in Figures 31.4, 31.8 and 31.9.

Proof. To create an implementation of an APL individual of type domain error (type 6), execute the assignment:

x :- new *d_error;*

By inspecting the declaration of individuals of this type, it is straightforward to verify that *x* has the attribute required of such an individual in Section 21.4. A similar argument applies to the remaining special individuals.

Lemma 6. In any program containing the declarations shown in Figures 31.4, 31.8 and 31.9, each member of *e_vector*, *d_error*, *r_error*, *l_error*,

i_error, *v_error* and *s_error* implements an APL individual.

Proof. When a member of these classes is created, it has the correct attributes and it is impossible to change these attributes. This statement can be verified by comparing the declaration of the definitions of the special individuals in Section 21.4.

In the implementation, there is a variable *empty_vector* whose value is an instance of class *e_vector* and this one instance is always used when an empty vector is required. Similarly, the value of the variable *domain_error* is an instance of class *d_error* and this one instance is always used when the individual *DOMAIN ERROR* is needed. The same statement applies to the variables *rank_error*, *length_error*, *index_error*, *value_error* and *syntax_error* with respect to classes *r_error*, *l_error*, *i_error*, *v_error* and *s_error* and the the corresponding individuals.

31.2.7 Labels

The declaration of *individual* class *apl_label* is shown in Figure 31.10. All APL individuals of type label (type 13) are required to have the same rank, shape, cardinality and type attributes and these values are included in the declaration. The single component of the elements attribute must be a positive integer. The parameter *e* is the elements attribute of instances of *apl_label* and the type checking mechanism of SIMULA forces this value to be a real representation of an integer. The if statement in the block terminates execution if a negative integer is passed as the parameter.

It is straightforward to prove the following lemmas:

Lemma 7. Each APL individual of type label (type 13) is implemented by an instance of class *individual* as declared in Figures 31.4 and 31.10.

Lemma 8. As long as any program containing the declaration of *individual* in Figures 31.4 and 31.10 is running without error termination, each member of *apl_label* implements an APL individual.

```
special class distinguished;
begin  end;

distinguished class d_error;
begin

   integer procedure type;
      type := 6;

end of d_error;

distinguished class r_error;
begin

   integer procedure type;
      type := 7;

end of r_error;

distinguished class l_error
begin

   integer procedure type;
      type := 8;

end of l_error

comment The declarations of the remaining
        distinguished elements are similar.;
```

Figure 31.9

```
individual class apl_label(e); value e; integer e;
begin

   integer procedure rank;
      rank := 0;

   integer procedure shape(i); value i; integer i;
      shape := 0;

   integer procedure cardinality;
      cardinality := 1;

   integer procedure type;
      type := 13;

   real procedure element(i);  value i; integer i;
      element := e;

   if e < 1
      then Abort("INDIVIDUAL: Invalid label.")

end of apl_label;
```

Figure 31.10

31.2.8 Function Definitions

The declaration of *individual* class *function_definition*, as used at this stage of the implementation is sketched in Figure 31.11. It can be shown that each member of this class satisfies the requirements for function definition geven in Section 26.2 and that each APL individual of type function definition that meets these requirements is implemented by a member of this class.

This class declaration is only used in Chapter III; it will be replaced by a more elaborate definition that is adequate for the full function definition mechanism of APL in Chapter V.' Therefore, a proof of the properties of this class is omitted; they are simply stated as propositions.

Proposition 9. Each APL individual of type function definition (type 12) is implemented by an instance of class *individual* as declared in Figures 31.4 and 31.11.

Proposition 10. As long as any program containing the declaration of *individual* in Figures 31.4 and 31.11 is running without termination, each

```
individual class function_definition(elements);
   value elements; real array elements;
begin

   integer procedure rank;
      rank := 1;

   integer procedure shape(i); value i; integer i;
      if i = 1
         then shape := 2
         else Abort("FN DEF: Shape subscript out of range.");

   integer procedure cardinality;
      cardinality := 2;

   integer procedure type;
      type := 12;

   real procedure element(i); value i; integer i;
      if i = 1  or i = 2
         then element := elements[i]
         else Abort("FN DEF: Element subscript out of range.");

   if not <elements contain function definition>
      then Abort("FN DEF: Creating invalid function definition.");

end of function_definition;
```

Figure 31.11

member of *function_definition* implements an APL individual.

31.2.9 Theorems

The following theorems assert that the present implementation of APL individuals is correct in the sense that each member of *individual* has the properties of an APL individual as long as the program is running without error termination. In the following sections, it will be shown that the conditions which cause error termination cannot arise. Hereafter, unless there is a need to make a careful distinction, it will be assumed that each *individual* in the implementation is the same as an APL individual in the formal semantics.

Theorem 1. Each APL individual whose rank and cardinality are less than or equal to *maxint* and such that each component of its shape is less than or equal to *maxint* is implemented by an instance of *individual* as declared in Figures 31.4 to 31.11.

Proof. By lemmas 3, 5 and 7 and proposition 9.

Theorem 2. As long as any program containing the declaration of *individual* in Figures 31.4 to 31.11 is running without error termination, each *individual* created using *apl_number, apl_boolean, apl_character, e_vector, d_error, r_error, ;_error, v_error, s_error, apl_label* or *function_definition* implements an APL individual.

Proof. By lemmas 4, 6 and 8 and proposition 10.

Corrolary. As long as any program containing the declaration of *individual* in Figures 31.4 to 31.11 is running without error termination, if there are no occurences of new applied to *individual, usual, numerical, special* or *distinguished*, then the value of any variable of type *individual* (that is, a ref*(individual)*) implements an APL individual.

31.2.10 Implementation Details

The declaration of *individual* used in the proof of theorems 1 and 2 is almost exactly the declaration used in the implementation. There are a few differences that make it easier to use individuals but these changes do not modify the proofs of the lemmas and theorems.

In the implementation, each *individual* has an additional procedure attribute *output* which writes the *individual* on the user's terminal in a format that closely resembles the output from the DEC and IBM implementations of APL. This procedure uses a procedure *output_single_element* that is declared in each class that is a leaf of the tree in Figure 31.2. It has been shown that these procedures have the properties claimed but the proof is not presented here because these procedures will be replaced in Chapter VI when format (▼) is defined.

The second change appears to be a fundamental change in the structure of *individual* but it is actually a significant simplification of the implementation. The *type* attribute of *individuals* is omitted in the implementation because it is not needed. The type attribute of an APL individual is used in tow places in the formal semantics: The type attribute is used to check for error conditions in the definitions of primitive functions and to set the type attribute of the value of a primitive functions.

The definitions of primitive functions contain occurrences of formulas of recursive arithmetic of the form:

type (*B*) *E* {5,6,...,11}

type (*B*) *E* {6,7,...,11}

type (*B*) *E* {1,2}

type (*B*) = 3

If *b* is an *individual*, then the following SIMULA boolean expressions have the same truth value (in the same order);

b in *special*

b in *distinguished*

b in *numerical*

b is *apl_character*

It is easy to see why this is the case. For example, consider the first expressions. By examining the declaration of *individual* it can be verified that the only *individuals* with a *type* attribute such that *5* ≤ *type* ≤ *11* are members of *special* or of *special* class *distinguished*. By the definition of in, the expression *b* in *special* is true just in the case that *b* is in *special* or *b* is in *distinguished*. This means that this expression is true exactly when *5* ≤ *type* ≤ *11* as required. Similarly, the only *individuals* with *type* = *3* are members of *apl-character*. By the definition of is, *b* is *apl_character* is true only if *b* is a member of *apl_character* and so the two expressions have the same truth value. A similar argument applies for the remaining expressions.

This use of subclasses provides a significant simplification. For example, when the formal semantics contains an expression of the form:

```
if type(B) = 1
   then ...
if type(B) = 2
   then ...
if type(B) =3
   then ...
...
```

a straightforward implementation of this test for an *individual b* is:

```
if b.type = 1
   then ... else
if b.type = 2
   then ... else
if b.type = 3
   then ... else
...
```

However, using subclasses, the definition is implemented as follows:

```
inspect b
   when apl_number do ...
   when apl_boolean do ...
   when apl_character do ...
```

This construction is much easier to understand and is equivalent to the first implementation.

By the definition of the inspect statement, the first when clause such that *b* is in the class named after when is executed and the remaining when clauses are skipped. This is exactly what happens when the nested if statement is executed.

By the definiton of the SIMULA inspect statement, the first when clause such that *b* is in the class named after the keyword when is executed and the remaining when clauses are skipped. This is exactly what happens during the execution of the nested if statement since the class names correspond to the type numbers. Therefore, the inspect statement is equivalent to the if statement.

The second place that the type attribute is used in the formal semantics is to set the type of the individual that is the value of a primitive function. This is in a context such as the following:

```
otherwise
   rank(Z) := shape(B)
   shape(Z) := shape(B)
   type(Z) := type(B)
```

In the implementation an array s containing shape(*B*) is created with a statement of the form

```
for i := 1 step 1 until b.rank do
   s[i]:= b.shape(i);
```

and then a new *individual* could be created with a statement of the form:

```
if b.type = 1
   then z :- new apl_number(b.rank,s) else
if b.type = 2
   then z :- new apl_boolean(b.rank,s) else
if b.type = 3
   then z :- new apl_character(b.rank,s);
```

By the above argument, a more readable and equivalent form of this statement is:

```
inspect b
   when apl_number do z :- new apl_number(rank,s)
   when apl_boolean do z :- new apl_boolean(rank,s)
   when apl_character do z :- new apl_character(rank,s);
```

[An otherwise clause is not needed here because the type of *b* is known to be one of these types when the statement is executed.]

Although a type procedure is not needed, it is straightforward to write such a procedure. Here is an outline of such a declaration:

```
integer procedure type(x); ref (individual) x;
inspect x
   when apl_number do type := 1
   when apl_boolean do type := 2
   when apl_character do type := 3
   when e_vector do type := 5
   ...
   when s_error do type := 11
   when function_definition do type := 12
   when apl_label do type := 13;
```

APPENDIX E
INTRODUCTION TO SIMULA

This Appendix provides a brief introduction to SIMULA-67 so that the present volume will be self-contained. Dahl [1] provides an excellent description of SIMULA and its application in many different programming problems.

SIMULA may be thought of as Algol-60 without own variables and with value as the default parameter passing mechanism augmented by the concept of a class. Classes are used to provide security for programs, to provide arbitrary data structures and coroutines as well as to provide for easy construction of simulations. The following discussion addresses those aspects of SIMULA that are used in this work.

E1. Classes

The basic idea of the class mechanism is that is provides a means for creating instances of blocks that remain in existence when control leaves the block and for naming such block instances as well as the procedures and variables that are inside such blocks.

A good starting point for explaining classes is procedures. The declaration of a procedure provides a template for creating a block instance of the procedure body. The local variables of the procedure are declared inside the procedure and the referents of global variables are determined by the location of the declaration in accord with Algol's static scope rules. When the procedure is called, the block that is the body of the procedure is created including storage allocation for the local variables and parameters, and the parameters are copied into the block instance in the appropriate way. After this, control moves to the first executable statement of the main block of the procedure. After the body of the procedure is executed, the block that is the body of the procedure is deleted. [The code still exists in the core image but it is inaccessible except by calling the procedure again.]

The declaration of a class is similar to the declaration of a procedure; it is a template for creating a block instance. Here is an example:

```
class Foo(x,y); value x,y; integer x; real y;
begin
   integer a,b;
   real c;

   integer procedure f(d); value d; integer d;
      begin ... end;

   a := x+y;
   b := x*4
   c := y**2
end
```

The variables x and y are formal parameters of the class in the same way that procedures have parameters. The attributes of *Foo* are the variables x, y, a, b and c as well as the procedure f.

A class declaration should also be thought of as a declaration of a data type and it is possible, and indeed necessary, to have variables of this type. For example, if *Foo* is declared as above, the following declaration are legal:

```
ref(Foo) alpha, beta;
ref(Foo) array gamma[1:n,1:m];
```

The range of these variables is blocks created in accord with the template provided by the declaration of *Foo*.

It is easiest to describe the creation of class instances by means of an example:

```
ref(Foo) alpha, beta;
alpha :-new Foo(3,4);
beta :-new Foo(1,2)
```

[The character :- is the assignment operator for ref variables.] When the assignment to *alpha* is executed, an instance of class *Foo* (that is, a block) is created. Storage is allocated for the variables and parameters. The parameters are passed and then the body of the block is executed. After the body is executed, the block remains in existence and the block is the value of *alpha*. Control then passes to the next assignment statement. In a similar way when the second assignment statement is executed a second instance of the block described by the declaration of *Foo* is created and becomes the value of *beta*.

These two blocks are different! In the block *alpha*, $x=3$, $y=4$, $a=7$, $b=12$ and $c=16$ while in block *beta* $x=1$, $y=2$, $a=3$, $b=1$ and $c=4$. In addition,

if any of the variables *x*, *y*, *a*, *b* or *c* are global variables inside procedure *f*, executing *f* from these two blocks might well give different results.

It is possible to refer to the attributes of both of these blocks. For example, *alpha.x* refers to the attribute *x* of block *alpha* and *beta.x* refers to the attribute *x* of block *beta*, e.g., *alpha.x=3* and *beta.x=1*, *alpha.a=7* and *beta.a=3*, etc. The procedure in these blocks is called in the usual way; here are examples:

w := alpha.f(i);

q := beta.f(i)

These blocks remain in existence until one of two things happens: (1) The block in which *alpha* and *beta* are declared is exited so the variables *alpha* and *beta* no longer exist. (2) The variables *alpha* or *beta* are assigned another value and there is no other variable that has the block as its value.

There is an empty block called none which has no attributes and which can be the value of any ref variable. For example, after the statement

alpha :- none

is executed *alpha* has the value none. If no other variable has the block that was the value of *alpha* as its value, then this block is destroyed. Likewise, if the statement

alpha :- *beta*

is executed, then the value of *alpha* and *beta* are the same block and the old value of *alpha* may be lost. If this statement is followed by

beta :- none

then *alpha* refers to the old value of *beta* and the empty block is the value of *beta*.

The symbols == and =/= are, respectively, the equality and the non-equality operators for ref variables. The value of the expression

alpha == *beta*

is true if *alpha* and *beta* refer to exactly the same block. Otherwise

alpha =/= *beta*

has the value true.

E2. Data Structures

The use of classes to construct and manipulate data structures is best

described by giving examples. As an example of a data structure built with classes, consider a binary tree. A class *Node* is declared as follows:

```
class Node(data); character data;
begin
   ref(Node) left_son, right_son;
   left_son :- none;
   right_son :- none
end
```

These *Node*s have three attributes: (1) a data object, (2) a left son pointer and (3) a right son pointer.

Suppose the tree shown in Figure E2.1 has been constructed and that this tree is the value of the ref*(Node)* variable *a*. Inserting a *Node* with data 'Z' as the left son of the *Node* marked by the arrow is accomplished by;

```
temp :- a;
while temp.left.son ≠ none do
   temp :- temp.left_son;
temp.left_son :- new node('Z')
```

The data attribute of the *Node* marked with the circled arrow is referred to by

a.left_son.right_son.data

(These examples assume that the value of *a* is the root of the tree.)

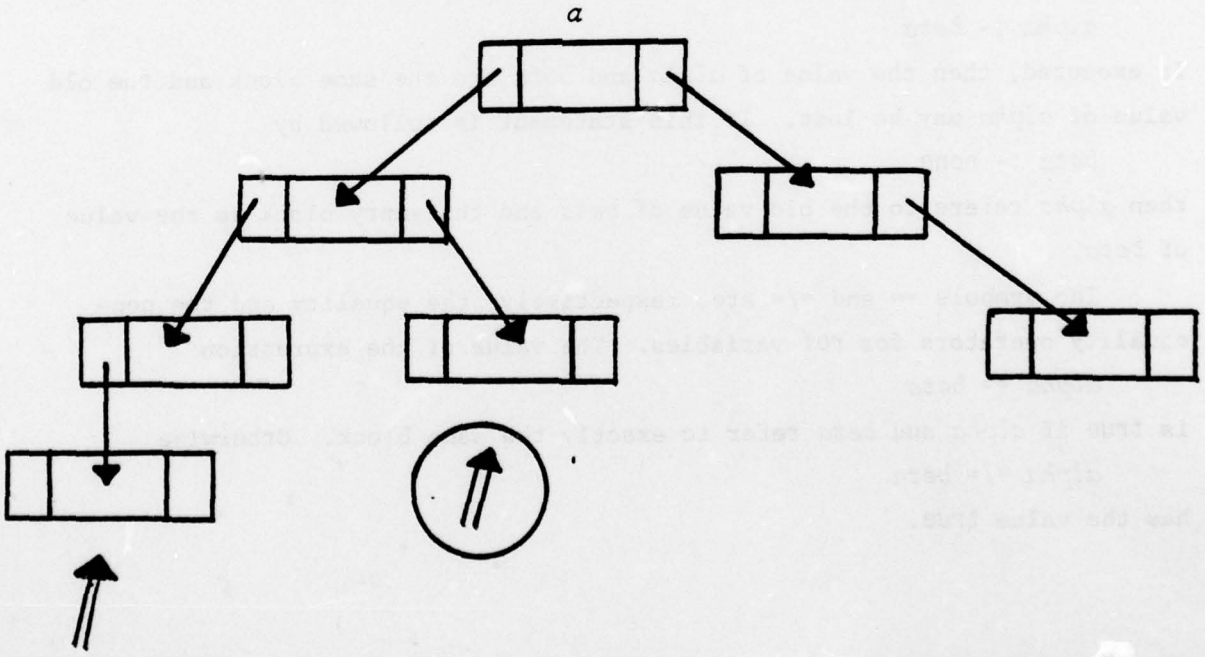


Figure E2.1

It is important to note that procedures can be attributes of *Nodes* in a data structure and it is not necessary for all *Nodes* of a data structure to have the same attributes. Suppose a type *node* which refers to both tree nodes and linked list nodes is needed and that both nodes must have a procedure attribute print that prints the data of a node of either kind. The following declarations meet these requirements

```
class Nodes(data); character data;
begin
   procedure print;
   begin
      outchar(data); outimage
   end
end;

Nodes class tree_node;
begin
   ref(Nodes) left_son, right_son;
   left_son :- none
   right_son :- none
end;

Nodes class List_node;
begin
   ref(Nodes) next;
   next :- none
end
```

As a result of these declarations, the attributes of a *Tree_node* are: *print*, *data*, *left_son* and *right_son* and the attributes of a *List_node* are *print*, *data* and *next*. Moreover, in spite of the restrictive typing of SIMULA, a *Tree_node* may be the value of *next* in a *List_node* and a *List_node* may be the value of *left_son* or *right_son* in a *Tree_node*.

Here are some examples of the use of these subclasses:

```
ref(Tree_node) alpha;
ref(List_node) beta, gamma;
alpha :- new Tree_node('a');
beta :- new List_node('b');
gamma :- new List_node('c');
```

These nodes can now be connected together:

alpha.left_son :- *beta;*

alpha.right_son :- *gamma;*

to give the data structure shown in Figure E2.2.

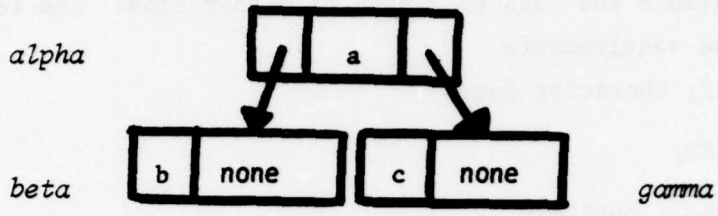


Figure E.2.2

Additional nodes could be added to the data structure by executing:

beta.next :- new *Tree_node*('d');

gamma.next :- new *List_node*('e');

to give the data structure in Figure E2.3.

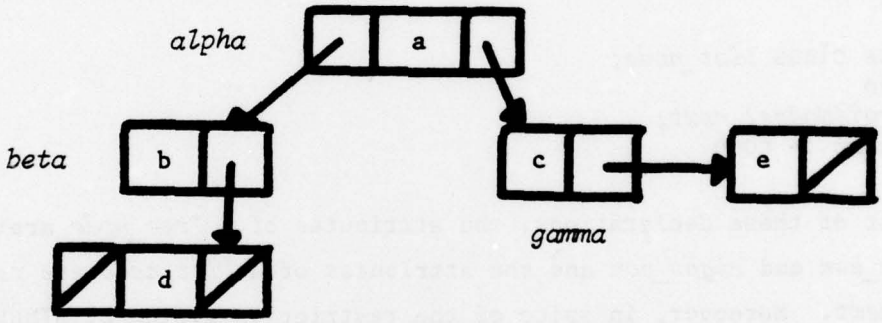


Figure E2.3

There are three points to be made about these examples.

(1) If a class is a sub-class of another class it has its own attributes as well as the attributes of its parent class(es).

(2) Elements can be arbitrarily mixed in a data structure.

(3) The procedures that are attributes of nodes of the data structure can be executed. For example, in the data structure of Figure E2.3, executing the procedure call

beta.print;

will write the character 'b' on the output listing using the procedure *print* that is an attribute of all *Node*s.

E3. Name Qualification

A number of operations and statements specifically designed to facilitate the use of classes and sub-classes are provided.

The relational operators is and in are used to determine the class membership of an object. If *x* is a ref variable and *Foo* is a class, then the expression

x is *Foo*

has the value true if the value of *x* is an instance of *Foo*, and the expression

x in *Foo*

has the value true if either *x* is *Foo* is true or if the value of *x* is an instance of a sub-class of *Foo*. Assuming the above declarations and assignments for *alpha*, *beta* and *gamma*, here is the truth table for is:

is	*Nodes*	*Tree-node*	*List-node*
alpha	false	true	false
beta	false	false	true
gamma	false	false	true

and the truth table for in:

in	*Nodes*	*Tree-node*	*List-node*
alpha	true	true	false
beta	true	false	true
gamma	true	false	true

The operator qua is used to tempotatily requalify a ref variable for the purpose of accessing sub-class attributes. Here is an example of the kind of problem that requires the use of requalification.

Suppose the declarations of *alpha*, *beta* and *gamma* in the above example had been

ref*(Nodes) alpha, beta, gamma;*

Variables declared in this way may have instances of the subclasses of *Nodes* as well as instances of *Nodes* itself as their values. Thus the subsequent assignments

```
alpha :- new Tree_node('a');
beta  :- new List_node('b');
gamma :- new List_node('c')
```

are all legal. The two assignments

```
alpha.left_son :- beta;
alpha.right_son :- gamma;
```

however, are illegal, as are the assignments

```
beta.next :- new Tree_node('d');
gamma.next :- new List_node('e');
```

The problem here is not with the actual assignment operation but with the attribute references

```
alpha.left_son
alpha.right_son
beta.next
gamma.next
```

Since *alpha*, *beta* and *gamma* were declared *Nodes*, they are only known to have the attributes data and print of *Nodes*, in spite of the fact that their actual values have additional attributes. Therefore, the expression

```
alpha.left_son
```

for example, results in a compilier error message to the effect that *left_son* is not an attribute of *Nodes*.

This situation is a result of the complete type and reference checks that are performed by the compiiier to increase program reliability and to increase relevance of error messages. There are, however, situations in which it is desirable and necessary to refer to sub-class attributes. The operator qua and the inspect statement provide these references.

As the example illustrated, the default qualification level of a reference variable is determined by its declaration. Thus the qualification levels of *alpha* is *Nodes*. In order to access the attributes *left_son* and *right_son*, however, the qualification level must be *Tree_node*. The operator qua is one way of achieving the desired qualification access level. The expression

```
alpha qua Tree_node
```

refers to the same object as *alpha* but at the *Tree_node* qualification level. Thus the expression

```
alpha qua Tree_node.left_son
```

is legal and refers to the desired attribute of *alpha*. Given the alternative *Nodes* declaration for *alpha*, *beta* and *gamma*, the four illegal assignments above may be legally written as follows:

```
alpha qua Tree_node.left_son :- beta;
alpha qua Tree_node.right_son :- gamma;
beta qua List_node.next :- new Tree_node('d');
gamma qua List_node.next :- new List_node('e');
```

The qua operator may be misused in two ways. One is if the qualified variable may not legally reference the qualification class. For example, with the declaration

```
ref(list_node) foo;
```

the expression

```
foo qua Tree_node
```

is illegal. Similarly, if the value of the qualified variable is not an instance of the qualification class or one of its subclasses, then the qualification is illegal. For example, after the assignment

```
alpha :- new Tree_node('a')
```

evaluating the expression

```
alpha qua List_node
```

results in a run time termination error.

Situations often arise where it is only known that the value of a variable *x* is an instance of one of some number of classes and it must be correctly qualified. For example, assume the declaration

```
ref(Nodes) x,y;
```

and suppose the value of *y* is to be assigned either to *right_son* of *x* if *x* is a *Tree_node* or to *next* of *x* if *x* is a *List_node*. If *x* is neither a

Tree_node nor a *List_node* then an error message is printed.

One solution, using both is and qua is the conditional statement:

```
if x is Tree_node
  then x qua Tree_node.right_son :- y else
if x is List_node
  then x qua List_node.next :- y else
outtext("Error");
```

An alternative way of specifing qualification levels, and one which is particularly useful in situations like this, is to use the inspect statement.

There are two general forms for the inspect statement. One is:

inspect <object-reference> do <stmt>

which executes the statement <stmt> as if it were textually within the body of the object denoted by <object-reference> (without violating protection). The primary advantage of this form of the inspect is that attributes of the <object-reference> may be accessed without the dot notation within <stmt>. The default qualification of <object-reference> is assumed within <stmt>.

The other form of the inspect statement is:

```
inspect <object-reference>
  when <class-name-1> do <stmt1>
  when <class-name-2> do <stmt2>
        .
        .
        .
  otherwise <stmt>
```

This statement behaves exactly like the construction

```
if < object-reference> in < class-name-1>
  then inspect < object-reference> qua < class-name-1>  do < stmt-1>  else
if < object-reference> in < class-name-2>
  then inspect < object-reference> qua < class-name-2> do < stmt> else
        .
        .
        .
  else < stmt>
```

The when clauses are examined in order until one is found such that < object-reference> in < class-name-?> is true. The statement in the do part of that when clause is then executed as if it were in the body of < object-reference> at the qualification level < class-name-?>. If < object-reference> is not in any of the classes then the statement following the otherwise is executed.

Using this second form of the inspect statement, the above conditional expression may be written much more clearly as;

```
inspect x
  when Tree_node do right_son :- y
  when List_node do next :- y
  otherwise outtext("Error");
```

E4. Protection

The second application of SIMULA classes is to provide protection for a group of procedures and variables. The usual Algol scope rules make it very difficult to insert a set of procedures and variables into someone else's Algol program. The result is complicated directions and no security at all. Typical directions for inserting into a program read something like this;

(1) Insert the following variable declarations in your main block declarations:

< list of variable declarations>

(2) Insert the following procedure declarations in your main block:

< list of procedure declarations>

(3) Insert the following executable statements before the first executable statement of your main block:

< list of initialization statements>

Users often make mistakes inserting the statements and there is absolutely no protection from their using any of my variables or using sub-procedures the wrong way. Further, they have the complete text before them.

Seperately compiled classes solve two of the problems completely. To make a set of procedures available to others, simply compile a class declared as follows:

```
class Foo
begin
  . <variable declarations>
  . <procedure declarations>
  . <initialization statements>
end
```

(This class declaration may not have global variables.) A user of these objects inserts the following at the beginning of their programs;

```
external class Foo;
ref (Foo) x;
x :- new Foo
```

The procedures and variables that are provided are now referenced by prededing their names with the string "*x*". This solves the problem of initializations, accidental references to internal variables and keeps the text of provided procedures confidential.

This prefixing of the string "*x*." may be avoided by qualifying the entire program as follows:

```
external class Foo;
inspect new Foo do
begin
  . <remainder of program>
end
```

It is possible to prevent references to variables and procedures outside the class. For example, if *Foo* is declared

```
class Foo;
protected a, b, c;
begin
  ...
end
```

then the attributes *a*, *b*, *c* cannot be referenced outside *Foo*. Similarly, if *Foo* is declared

```
class Foo
not protected a, b, c;
begin
  ...
end
```

then the only attributes that can be referenced outside a class instance are *a*, *b*, and *c*. Further, if a variable is protected, read only access can be provided with a dummy procedure;

```
class Foo;
protected
begin
  integer aa;
  integer procedure a;
      a := aa;
end
```

Similarly, references to variables and procedures of a class by a subclass can be protected using the keyword hidden as protected was used above. As a routine security precaution, it is helpful to declare everything that is not known to be needed elsewhere to both hidden and protected. This has saved much grief as errors are caught by the compiler or the runtime system rather than by tracing some weird bug.

This mechanism works very well. The only disadvantage is that all of the external references to attributes of a class are qualified names instead of simple names. The availability of the inspect qualification of blocks reduces the complexity of qualified references but a more general mechanism for renaming would be helpful. Hoare describes it abstractly in [2].

E5. Coroutines

The third use of SIMULA classes is for the construction of coroutines. In order to use classes for coroutines, it is necessary to associate a location counter with each class instance or block and to distinguish between two kinds of class instances. If the location counter of a class instance is after the last statement of the block the class instance is said to be terminated and otherwise it is active.

There are three primitives (Call, Detach, Resume) for terminating execution of a class instance before the last statement of the main block is ex-

ecuted and transferring the processor to another place. The three differ in the destination of the processor and in the way that responsibility for returning to a higher level procedure or class instance is treated.

At the time when the statement

Call(x);

is executed the value of x must be an active class instance. When it is executed, control passes to class instance x and execution continues using the location counter of x. In addition, when x terminates execution by either completing execution of the last statement in the main block or by executing a Detach, control returns to the statement following the Call.

At the time when the statement

Resume(x);

is executed, the value of x must be an active class instance. This statement can only be executed in a class instance; it may not appear in the main body of a program. When the statement is executed the following happens: (1) The location counter of the class instance in which the resume is executed is set to the statement after the Resume. (2) Control is transferred to class instance x and execution continues using the location counter of x. (3) When execution of x is terminated (by executing the last statement of x or by executing a Detach in x) control returns to the class that initiated the execution of the class containing the Resume. That is, executing a Resume transfers the responsibility for returning to its caller to x.

The statement

Detach;

may only be executed in a class instance. Two things happen when it is executed. (1) Control leaves the class instance in which the Detach is executed and its location counter is set to the statement following the Detach. (2) Control is returned to the caller of the class instance (or to the inherited caller).

An example may help.

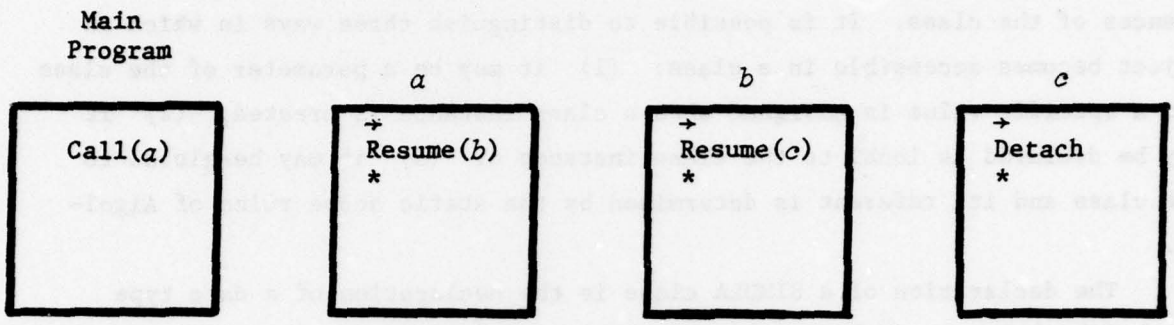


Figure E4.1

When the call in the main program is executed, control passes to *a* at its location counter (the arrow) and *a* has the obligation of returning to the main program. When the resume in *a* is executed, its location counter is set to the statement after the resume (the *). Execution of *b* begins at its location counter and *b* has inherited *a*'s obligation to return to the main program. Similar things happen when *b* executes its resume. Execution of *c* begins at its location counter and *c* has inherited the obligation to return to the main program. When *c* executes its detach, control passes to the statement after the call in the main program. Of course, the object labeled main program may itself be a class instance.

The principal restriction associated with the use of classes as coroutines is the restriction on context. The referants of global variables in a class declaration are determined by the location of the class declaration in the text of the program. Thus, it is not possible to have a dynamic environment without passing the environment to the class instance as a parameter of the class. This restriction is directly related to the block structure of Algol.

The earlier discussion of creating class instances may suggest that new cannot be used to create an active class instance; this is not the case! When a class instance is created execution of the main block is started. It may be terminated by executing a Detach and when this Detach is executed the location counter is set to the statement after the Detach and execution of the new is terminated.

In SIMULA one distinguishes between the declaration of a class and instances of the class. The declaration serves as a template for creating instances of the class. It is possible to distinguish three ways in which an object becomes accessible in a class: (1) it may be a parameter of the class and a specific value is assigned when a class instance is created, (2) it may be declared as local to the class instance or (3) it may be global to the class and its referent is determined by the static scope rules of Algol-60.

The declaration of a SIMULA class is the declaration of a data type (just like integer, real, etc.) and it is possible to declare variables whose range is objects of this type. These variables may be part of compound names that are used to refer to objects and attributes of objects. The only restriction on the length of compound names is that an identifier must fit on a single input line. The power of this naming convention is weakly suggested by the examples given above.

Each class instance has its own storage for data and a location counter and, of course, to the code that is the body of the class and its internal procedures. Execution of a block may be terminated by control reaching the end of the block instance, or by executing a Detach or a Resume and it may be continued at a later time by means of a Call or a Resume executed in another block instance.

REFERENCES

[1] O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare. Structured Programming. London and New York, Academic Press, 1972.

[2] C.A.R. Hoare. Proof of Correctness of Data Representations. Acta Informatica, Vol. 1 (1972), pp. 271-281.